University of Zurich[UZH]

Department of Informatics

Leveraging Sort-Merge for Processing Temporal Data

A dissertation submitted to the Faculty of Economics, Business Administration and Information Technology
of the University of Zurich

for the degree of
Doctor of Science (Ph.D.)

by

Francesco Cafagna

from Italy

Accepted on the recommendation of

Prof. Dr. Michael H. Böhlen
Prof. Dr. Sherif Sakr

2016

University of Zurich[UZH]

The Faculty of Economics, Business Administration and Information Technology of the University of Zurich herewith permits the publication of the aforementioned dissertation without expressing any opinion on the views contained therein.

Zurich, September 21, 2016

Head of the Ph.D. committee for informatics: Prof. Dr. Elaine M. Huang

Abstract

Sorting is, together with partitioning and indexing, one of the core paradigms on which current Database Management System implementations base their query processing. It can be applied to efficiently compute joins, anti-joins, nearest neighbour joins (NNJs), aggregations, etc. It is efficient since, after the sorting, it makes one sequential scan of both inputs, and does not fetch redundantly tuples that do not appear in the result.

However, sort-based approaches loose their efficiency in the presence of temporal data: *i)* when dealing with time intervals, backtracking to previously scanned tuples that are still valid refetches in vain also tuples that are not anymore valid and will not appear in the result; *ii)* when dealing with timestamps, in computing NNJs with grouping attributes, blocks storing tuples of different groups are refetched multiple times. The goal of this thesis is to provide support to database systems for performing efficient sort-merge computations in the above cases.

We first introduce a new operator for computing NNJ queries with integrated support of grouping attributes and selection predicates. Its evaluation tree avoids *false hits* and *redundant fetches*, which are major performance bottlenecks in current NNJ solutions. We then show that, in contrast to current solutions that are not group- and selection-enabled, our approach does not constrain the scope of the query optimizer: query trees using our solution can take advantage of any optimization based on the groups, and any optimization on the selection predicates. For example, with our approach the Database Management System can use a *sorted index scan* for fetching at once all the blocks of the fact table storing tuples with the groups of the outer re-

lation and, thus, reducing the tuples to sort. With Lateral NNJs, instead, groups are processed individually, and blocks storing tuples of different groups are fetched multiple times. With our approach the selection can be pushed down before the join if it is selective, or evaluated on the fly while computing the join if it's not. With an indexed NNJ, instead, selection push down causes a nested loop which makes the NNJ inefficient due to the quadratic number of pairs checked. We applied our findings and implemented our approach into the kernel of the open source database system PostgreSQL.

We then introduce a novel partitioning technique, namely Disjoint Interval Partitioning ($\mathcal{DIP}$), for efficiently computing sort-merge computations on interval data. While current partitioning techniques try to place tuples with similar intervals into the same partitions, $\mathcal{DIP}$ does exactly the opposite: it puts tuples that do not overlap into the same partitions. This yields more merges between partitions but each of those no longer requires a nested-loop but can be performed more efficiently using sort-merge. Since $\mathcal{DIP}$ outputs the partitions with their elements already sorted, applying a temporal operator to two $\mathcal{DIP}$ partitions is performed in linear time, in contrast to the quadratic time of the state of the art solutions. We illustrate the generality of our approach by describing the implementation of three basic database operators: join, anti-join, and aggregation.

Extensive analytical evaluations confirm the efficiency of the solutions presented in this thesis. We experimentally compare our solutions to the state of the art approaches using real-world and synthetic temporal data.

Zusammenfassung

Die Sortierung ist, zusammen mit der Partitionierung und der Indexierung, eines der Kernparadigmen, auf der die Verarbeitung von Anfragen durch Datanbanksysteme beruht. Sie wird unter anderem für die effiziente Berechnung von Joins, Anti-Joins, Nearest Neighbour Joins (NNJs) und Aggregationen angewandt. Die Effizienz der Sortierung rührt daher, dass nach ihr lediglich ein sequenzieller Scan zweier sortierter Relationen für die Beantwortung der eingangs erwähnten Anfragen durchgeführt werden muss und auf Tupel, welche nicht Bestandteil des Ergebnisses sind, nicht mehrfach zugegriffen wird.

Allerdings verlieren Ansätze, die auf der Sortierung basieren, ihre Effizienz bei Anfragen über zeitabhängigen Daten: *i)* bei Zeitintervallen wird beim Zurückgreifen auf vorgängig zugegriffene und immer noch gültige Tupel erneut auf inzwischen ungültige und in der Ergebnismenge nicht enthaltene Tupel zugegriffen; *ii)* bei Zeitpunkten wird bei der Berechnung von NNJs mit Attributgruppierung auf Blöcke mit Tupeln verschiedener Gruppen mehrfach zugegriffen. Das Ziel dieser Arbeit besteht in der Weiterentwicklung von Datenbanksystemen hinsichtlich der effizienten Verarbeitung von Sort-Merge-Berechnungen in den obengenannten Fällen.

Zuerst stellen wir einen neuen Operator für die Berechnung von NNJ-Abfragen mit integrierter Unterstützung von Attributgruppierung und Auswahlprädikaten vor. Sein Evaluationsbaum vermeidet *erfolglose* und *redundante Zugriffe* auf Daten, welche die hauptsächlichen Engpässe in der Performanz von aktuellen NNJ-Lösungen darstellen. Wir zeigen, dass im Gegensatz zu herkömmlichen Lösungen, die keine Attributgruppierungen und Auswahlprädikate unterstützen,

unser Ansatz die Möglichkeiten des Abfrageoptimierers signifikant erweitert: Abfragebäume, die unseren Ansatz anwenden, profitieren von sämtlichen Optimierungen aus dem Einsatz von Attributgruppierungen und Auswahlprädikaten. Beispielsweise können Datenbanksysteme mit unserem Ansatz einen sortierten Indexscan einsetzen, der genau einmal auf einen Block der Faktentabelle zugreift, der Tupel mit den Gruppen der äusseren Relation speichert, und dadurch die Anzahl der zu sortierenden Tupel verringert. Im Unterschied dazu werden mit gängigen lateralen NNJs die Gruppen einzeln verarbeitet und auf Blöcke, die Tupel verschiedener Gruppen beinhalten, wird mehrmals zugegriffen. Mit unserem Ansatz kann die Selektion bereits vor dem Join ausgewertet werden, sofern sie selektiv ist, oder die Selektion kann während des Scans der Daten ausgwertet werden. Mit einem indexierten NNJ führt eine standardmässig frühe Auswertung der Selektionsbedingung (selection push down) zu einer geschachtelten Schleife (nested loop), was den NNJ auf Grund der quadratischen Anzahl zu prüfender Paare ineffizient macht. Wir haben die gewonnenen Erkenntnisse angewandt und unseren Ansatz im Kern des Open Source Datenbanksystems PostgreSQL umgesetzt.

Wir führen eine neue Art der Partitionierung, nämlich Disjoint Interval Partitioning ($\mathcal{DIP}$), zur effizienten Verarbeitung von Sort-Merge-Berechnungen auf Intervalldaten ein. Aktuelle Ansätze zur Partitionierung versuchen Tupel mit ähnlichen Intervallen in dieselbe Partition zu packen. Unser Ansatz macht genau das Gegenteil: er weist nicht-überlappende Tupel denselben Partitionen zu. Dies führt zu mehr Kombinationen von Partitionen, aber da jede dieser Kombinationen keine geschachtelte Schleife erfordert, können Sort-Merge-Berechnungen effizienter durchgeführt werden. Da $\mathcal{DIP}$ die Elemente bereits sortiert ausgibt, kann ein Operator auf zwei $\mathcal{DIP}$-Partitionen in linearer Zeit durchgeführt werden, im Unterschied zur quadratischen Zeit herkömmlicher Lösungen. Wir zeigen die Allgemeingültigkeit unseres Ansatzes auf, indem wir die Umsetzung von drei Datenbankoperatoren beschreiben: Join, Anti-Join und Aggregation.

Umfassende analytische Auswertungen bestätigen die Effizienz der in dieser Arbeit vorgestellten Lösungen. Wir vergleichen unsere Lösungen mit aktuellen Ansätzen mit echten und synthetischen zeitabhängigen Daten.

Dedicated with deepest admiration to Gino Strada, Tiziana Rosa, Stefano Truzzi.
With your altruism you made me a better man.

Acknowledgments

I would like to thank my advisor Prof. Michael Böhlen for his guidance and encouragement during my PhD studies, for providing me insightful feedbacks during our meetings, and for having taught me to dig up the source of the problems.

I would like to thank the SNF and Agroscope for financing my research, and Annelies Bracher for introducing me to the real world problems solved during my PhD.

A special thank to Prof. Sherif Sakr for agreeing to co-advise this thesis and to Prof. Renato Pajarola for chairing the thesis defense.

I would also like to thank all my colleagues and former colleagues from the Database Technology Group at the University of Zurich for their help, discussions, and friendship.

Francesco Cafagna
Zurich, August 2016

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

1.1 Background

The problems studied in this PhD thesis are real world problems that we encountered during the development of the Swiss Feed Data Warehouse. The Swiss Feed Data Warehouse is a temporal Data Warehouse that we have built in tight collaboration with *Agroscope*, the Swiss Federal organization for agriculture, food and environmental research. The Swiss Feed Data Warehouse stores the nutritive contents of animal feed that is grown in Switzerland together with selected feeds imported from abroad. It contains the history over 40 years for 300 nutrients and more than 1000 animal feed types. Multiple temporal information are stored in the Swiss Feed Data Warehouse, such as, the anaylis time, sampling time, harvesting time, etc. Currently, almost 10 million measurements are stored in the Swiss Feed Data Warehouse. The Swiss Feed Data Warehouse is used by Swiss feed mills, research institutions, companies, and private farmers to compose healthy, effective and cheap animal feeding, and to optimize data collection and lab analyses. The system [BBB+12], [TBBC12] that we have built on top of the Swiss Feed Data Warehouse (available at http://www.feedbase.ch) offers a wide range of functionalities: it compares the temporal evolution of different nutrients in the animal feeds, it computes the cor-

relation between nutrients, it identifies regions in Switzerland where feeds with similar nutritive contents grow, and it computes and displays statistical information.

The concrete problem we have focused on in the first part of this PhD thesis is the computation of *derived nutrients*. Derived nutrients represent nutritive values that have not been physically measured in the lab (e.g., because of financial budget constraints) and are therefore calculated. A derived nutrient is defined and calculated through a chemical formula that refers to other nutritive values (e.g., Crude Protein, Vitamin D, etc.). If for a sample one or more nutrients that appear in the formula have not been determined in the lab, the closest measurements in time that are available for another sample with the same characteristics (e.g., it must be the same feed that was grown in the same canton at the same altitude) are considered. In the database literature, such operations are known as *Nearest Neighbour Joins* (NNJ). Our approach is the first NNJ solution that offers efficient support for *multiple groups* and *selective predicates* in NNJ queries. The approach remains stable when the number of animal feeds (i.e., the number of groups) on which a derived nutrient must be computed grows, and when many different nutrients are stored in the Data Warehouse (i.e., the predicate that selects the needed nutrient becomes selective).

In the second part of this PhD thesis we focused on the problem of computing database operators (such as joins, anti-joins, and aggregations) on the time intervals stored in the Swiss Feed Data Warehouse. For example, the interval $T = [T_s, T_e)$ in the fact table of Figure 1.1 expresses the period during which the animal feed has been stored (e.g., in a fridge or silos), i.e., the interval from the harvesting day T_s until the day T_e the feed has been given to the animals. We have developed a solution based on partitioning to efficiently compute temporal joins, anti-joins, and aggregations using sort-merge. Our solution is robust in the presence of long data histories since it minimizes the number of tuple comparisons and fetches the tuples sequentially, i.e., without random accesses. The Swiss Feed Data Warehouse is a good representative example since it stores a data history of 40 years.

In the third part of this PhD thesis we describe the architecture, interface and functionalities of the system that we have built on top of the Swiss Feed Data Warehouse. As an example, consider *data-driven menus*, i.e., menus whose selectable options depend on the previous selections made by the user. Since the data cube of the Swiss Feed Data Warehouse is sparse and for many combinations of dimension keys no data exists (e.g., feed Pea is not grown in canton Zurich; nutrient Phosphorus has never been measured on feed Oat Flakes; etc.), our system uses data-driven menus for providing the user with only the options for which data exist according to his/her previous selections. We have introduced an approach based on *partial evaluation* for

computing those menus efficiently. We conclude by describing three use cases of our system: as of July 2016, 3000 feed mills, research institutions, companies, and private farmers are using our system for composing healthy, effective and cheap animal feeding.

1.2 Sort-Merge Computations

Sorting is, together with partitioning and indexing, one of the core paradigms on which current DBMS implementations base their query processing. It can be applied to efficiently compute joins, anti-joins, nearest neighbour joins, aggregations, etc. Sort-merge computations are efficient since, after the sorting, they compute only one sequential scan of both inputs and do not refetch the tuples redundantly (i.e., tuples that do not appear in the result are not fetched more than once). We show this in the example below.

$\boldsymbol{R}$

	G	T_s	T_e
r_0	Pea	15	16
r_1	Pea	26	27
r_2	Pea	28	29
r_3	Soy	20	21

$\boldsymbol{S}$

	G	T_s	T_e	A	R	N	M
s_0	Pea	15	30	1030	0.9	CP	1.40
s_1	Pea	20	21	1000	1.0	CP	1.08
s_2	Pea	25	26	1020	0.5	OM	0.93
s_3	Pea	28	29	1110	0.9	CP	1.23
s_4	Soy	20	21	1000	0.8	CP	4.20
s_5	Soy	20	30	1000	0.3	OM	7.13
s_6	Soy	21	22	1100	0.9	CP	4.03
s_7	Whay	19	22	1000	0.8	CP	0.32

Figure 1.1: Outer Relation $\boldsymbol{R}$; Fact Table $\boldsymbol{S}$ with Lab Analyses on Feeds during June 2015.

Example 1. Consider our example relations where $\boldsymbol{R}$ are the query points and $\boldsymbol{S}$ is the fact table of the Swiss Feed Data Warehouse. We compute an equijoin between $\boldsymbol{R}$ and $\boldsymbol{S}$, i.e., $\boldsymbol{R} \bowtie_{G,T_s} \boldsymbol{S}$. The expression joins each $\boldsymbol{R}$ tuple with the tuples in $\boldsymbol{S}$ having the same values for the feed G and the harvesting time T_s. SortMerge sorts the relations as in the Figure, i.e., by (G,T_s), and then scans them. Tuple r_0 is compared with s_0 and, since they have the same feed and timestamp, they are output as join match. $\boldsymbol{S}$ is advanced because another tuple with the same feed and time might exist for r_0. However, this is not the case. Since $s_1.T_s > r_0.T_s$, $\boldsymbol{R}$ is advanced and r_1 is fetched and compared to s_1. This comparison does not produce a result tuple and, since $r_1.T_s > s_1.T_s$, $\boldsymbol{S}$ is advanced (first fetching s_2, then s_3). No match for r_1 exists. $\boldsymbol{R}$ is then advanced and r_2

produces a join match with s_3. The procedure continues similarly until all tuples are processed, building the result shown in Figure 1.2.

$\Pi_{G,T_s,N,M}(\boldsymbol{R} \bowtie_{G,T_s} \boldsymbol{S})$

	G	T_s	N	M
$r_0 \circ s_0$	Pea	15	CP	1.40
$r_2 \circ s_3$	Pea	28	CP	0.93
$r_3 \circ s_4$	Soy	20	CP	4.03
$r_3 \circ s_5$	Soy	20	OM	7.13

Figure 1.2: Equijoin Result.

In the version of the Swiss Feed Data Warehouse prior to this thesis, equijoins have been used for computing derived nutrients. *Derived nutrients* calculate nutritive values that have not been analysed in the lab. A *derived nutrient* (e.g., *Gross Energy*) is computed by evaluating a formula (e.g., *0.8* 'CP' + 2* 'OM'*) on the values of other nutrients (e.g, Crude Protein 'CP' and Organic Matter 'OM'). Equijoins were used to identify, for each query point, the nutritive values on which the formulas had to be evaluated.

Example 2. In this example we show the procedure for retrieving the nutritive values needed for computing the derived nutrient 'GE' (*Gross Energy*) for the query points in $\boldsymbol{R}$. Since 'GE' is computed by the formula *0.8* CP + 2* OM*, two equijoins are computed: first an equijoin using the 'CP' measurements, then one using the 'OM' measurements. Once for each query point its 'CP' and 'OM' measurements have been retrieved, the formula *0.8* CP + 2* OM* can be used for computing the 'GE' value.

$\Pi_{G,T_s,M/CP}(\boldsymbol{R} \bowtie_{G,T_s} \sigma_{N=\text{'CP'}}(\boldsymbol{S}))$

	G	T_s	CP
$r_0 \circ s_0$	Pea	15	1.40
$r_2 \circ s_3$	Pea	28	0.93
$r_3 \circ s_4$	Soy	20	4.03

$\Pi_{G,T_s,M/OM}(\boldsymbol{R} \bowtie_{G,T_s} \sigma_{N=\text{'OM'}}(\boldsymbol{S}))$

	G	T_s	OM
$r_3 \circ s_5$	Soy	20	7.13

Figure 1.3: Derived nutrient 'GE' computed using equijoins: r_3 is the only query point for which both a 'CP' and a 'OM' measurement exists.

As shown in the above example, for some tuples in $\boldsymbol{R}$, a 'CP' and/or a 'OM' measurement might not be available in the fact table of the Swiss Feed Data Warehouse. For example, r_3 is the only query point for which both a 'CP' and an 'OM' measurement is available in the fact table for the same feed and time. In other words, r_3 is the only query point for which the derived nutrient 'GE' can be computed using equijoins. This is so since nutrients are not measured on a daily basis in the Swiss Feed Data Warehouse (e.g., because of the costs of the lab analyses). To deal with this problem, Nearest Neighbour Joins are used by domain experts to find, for each query point, the temporal closest measurements available and using them for computing derived nutrients.

1.2.1 Inefficiencies of Sort-Merge in NNJs with Grouping Attributes

A Nearest Neighbour Join (NNJ) solution based on Sort-Merge has been proposed by Y. Silva [SAA10]. It efficiently computes $\boldsymbol{R} \ltimes^{T} \boldsymbol{S}$, i.e., for each $r \in \boldsymbol{R}$ it finds in $\boldsymbol{S}$ the tuple with the closest T value. Such a solution can be leveraged to manage tuples of different groups (i.e., avoiding that tuples of different groups are joined together) using the SegmentApply operator [GLJ01]. However such a solution suffers when the fact table is not clustered on the grouping attribute G, which is specified at query time. Consider our running example. To find the closest measurement available for a tuple $r \in \boldsymbol{R}$, we have to consider only the tuples that have the same feed value (e.g., for r_3 we want a value for 'Soy'and not simply the closest measurement on time (e.g., s_1) since it refers to a different animal feed). This is what the SegmentApply operator does. On our example, SegmentApply first fetches from $\boldsymbol{R}$ and $\boldsymbol{S}$ the tuples of feed 'Pea' (and runs a sort-merge NNJ), and then does the same for feed 'Soy'. Such a solution suffers, however, from *redundant fetches* since, if a block stores a tuple of group 'Soy' and a tuple of group 'Pea', such a block is fetched twice: once for computing the first NNJ, once for the second. The solution that we present in this thesis is *group-enabled* and fetches each block of the input relations at most once.

1.2.2 Inefficiencies of Sort-Merge with Time Intervals

Interval data is data that is associated with an interval $T = [T_s, T_e)$, where T_s is the (inclusive) starting point and T_e the (exclusive) ending point of the interval. In our example relation in Figure 1.1, T_s is the *harvesting time* of a feed, while T_e is the *feeding time*. Thus, T indicates the time interval during which the feed is stored (e.g., in silos) prior to its use.

The main problem of computing sort-merge computations over interval data is that, after the sorting, tuples whose timestamp overlaps a given query point are not consecutive but are distributed between other (non-overlapping) tuples. In other words, no total order exists for interval data. Sort-Merge can be leveraged for dealing with interval data but it is forced to backtrack during the computation (i.e., going back to previously scanned tuples) similar to sort-merge computations over non-key attributes. The main difference is however that, in the presence of interval data, many non-matching tuples might have to be rescanned, too, causing up to a quadratic number of unnecessary comparisons.

Example 3. Again, consider relations $\boldsymbol{R}$ and $\boldsymbol{S}$ in Figure 1.1. For simplicity, we consider only the green section of the relations, i.e., the tuples of feed 'Pea'. We want to find, for each tuple in $\boldsymbol{R}$, the tuples in $\boldsymbol{S}$ with an overlapping interval $T = [T_s, T_e)$. As shown, both relations are sorted by the starting point T_s of interval T. The relations are scanned concurrently. First, tuple r_0 is compared with s_0. The tuples are joined because $[15, 16)$ overlaps with $[15, 30)$. We proceed with tuples from $\boldsymbol{S}$ until we fetch a tuple starting after r_0 ends: only then we can be sure of having found all tuples overlapping r_0. Thus, no tuples after s_1 must be looked at. Next, tuple r_1 is fetched and we must *backtrack* in $\boldsymbol{S}$. To ensure that all join matches for r_1 are found, we must go back to the first join match of r_0 (i.e., s_0). Indeed, tuple r_1 overlaps with s_0. Tuple r_1 then is compared with s_1, s_2, and s_3. All those tuples in $\boldsymbol{S}$ are scanned *unproductively*, i.e., without producing a join match with r_1. The same procedure is applied for r_2, which overlaps with s_0 and s_3 but does not overlap with s_1 and s_2.

In the presence of just one tuple with a long interval (such as s_0), sort-merge makes a quadratic number of comparisons because of backtracking. This is inefficient since between two overlapping tuples many non-overlapping are rescanned, too. In this thesis we propose $\mathcal{DIP}$ (Disjoint Interval Partitioning) as a technique for computing sort-merge computations on interval data without backtracking, and thus reducing the number of unproductive comparisons done.

1.3 Contributions

This thesis makes three main contributions to the database field:

- It introduces a group- and selection-enabled Nearest Neighbour Join $\boldsymbol{R} \ltimes^{T} [\mathbf{G}, \theta] \boldsymbol{S}$, i.e., a Nearest Neighbour Join with similarity on T, and integrated support for grouping attributes

$\mathbf{G}$ and selection predicates θ. Our group- and selection-enabled NNJ is independent of the physical layout of the database and, opposite to the state of the art solutions, does not suffer from *redundant fetches* and *index false hits* independent of the number of groups and of the selectivity of the predicate. Ours is the first NNJ solution that does not restrict the scope of the query optimizer: while current solutions avoid to push-down the selection on θ or use an index on $\mathbf{G}$ with limited optimization possibilities, our approach supports any optimization rule on the selection predicate and on the grouping attribute.

- It introduces Disjoint Interval Partitioning $\mathcal{DIP}$, i.e., a new partitioning technique for computing joins, anti-joins and aggregation on interval data. $\mathcal{DIP}$ allows to make sort-merge computations on interval data limiting the number of unproductive comparisons per tuple to the number of partitions. Unlike state of the art solutions that also leverage sort-merge, $\mathcal{DIP}$ does not perform random accesses.

- It illustrates the architecture, interface and functionalities of a system that we have built on top of the Swiss Feed Data Warehouse and that it is used by 3000 farmers, scientists, feed mills, etc. We describe the lessons learned in implementing the core functionalities of our data warehouse based system; for example, we show how to apply partial evaluation to minimize the response time for creating data-driven menus. We finally describe three use cases of our system.

Each part of this thesis starts out with a problem of the animal feeding industry, followed by an analysis and precise definition of the problem. The solution to the problem and its properties are studied and elaborated analytically and then implemented. Large parts of this thesis have been integrated into the open source database system PostgreSQL and are used by the user as part of the system available at `www.feedbase.ch`. The implementation is extensively evaluated and compared with state-of-the-art approaches to confirm the analytical results of the solution.

The rest of this section elaborates the contributions of this thesis in more detail.

1.3.1 Nearest Neighbour Join with Groups and Predicates

The first contribution of this thesis is a group- and selection-enabled Nearest Neighbour Join operator $\boldsymbol{R} \ltimes^{T} [\mathbf{G}, \theta] \boldsymbol{S}$. Our operator offers robust support for grouping attributes and selection predicates in NNJ queries. It does not suffer from index false hits or redundant fetches, which

are major performance bottlenecks in current NNJ solutions that are not group- and selection-enabled.

Example 4. Consider our example relations where $\boldsymbol{R}$ and $\boldsymbol{S}$ are selections on the fact table of the Swiss Feed Data Warehouse. The operation we want to perform is a NNJ where as a grouping attribute we select the animal feed G, and as a selection predicate we use the nutrient to find, e.g., $\theta \equiv N =$ 'CP'. The result relation $\boldsymbol{Z}$ is shown in Figure 1.4. For example, the nearest neighbour of tuple r_0 is tuple s_0 since it is the closest 'CP' measurement for the same feed. For tuple r_1, s_3 is its nearest neighbour. Even if temporally closer, s_2 has not been chosen as nearest neighbour since it does not satisfy $N =$ 'CP'. Note that for tuple r_3, two measurements s_4 and s_6 for Soy exist with the same minimum distance and satisfying $N =$ 'CP', and therefore two join matches are returned.

$\mathbf{Z} = \boldsymbol{R} \ltimes^{T_S}[G, N =$ 'CP' $\wedge R > 0.7]\, \boldsymbol{S}$

	G	T	A	R	N	C
$r_0 \circ s_0$	Pea	15	1030	0.9	CP	1.40
$r_1 \circ s_1$	Pea	26	1100	0.9	CP	1.23
$r_2 \circ s_3$	Pea	28	1100	0.9	CP	1.23
$r_3 \circ s_4$	Soy	20	1000	0.8	CP	4.20
$r_3 \circ s_6$	Soy	20	1100	0.9	CP	4.03

Figure 1.4: NNJ Result **Z**.

A key feature of our approach is that, in contrast to all others NNJ solutions, it does not limit the scope of the query optimizer. In fact, depending on the group and predicate selectivities, we apply different query optimizations to our query evaluation tree. For example, in typical data warehouse scenarios, due to the high number and nature of the facts stored in the fact table, the group and predicate selectivities are low. Thus, the groups of the query points can be used to limit the fact table to its relevant portions. Solutions that are not group-enabled must process each group independently, and end up fetching (from disk or memory) the blocks of the fact table multiple times. Our approach fetches each block at most once independent of the group selectivity, and it performs well independent of whether the DBMS pushes the evaluation of the selection predicate down or evaluates it on the fly. In contrast, solutions that are not selection-enabled suffer from index false hits if selection push-down is not applied or end up in a nested loop if it is. The query tree that our approach produces in a DW scenario is the following:

1. Fetch $\boldsymbol{R}$
2. Fetch with an index all block IDs of $\boldsymbol{S}$ storing tuples with the groups of $\boldsymbol{R}$
3. Deduplicate the IDs
4. Fetch the blocks of $\boldsymbol{S}$ and evaluate θ on the fly
5. Sort the tuples by ($\mathbf{G}$,T)
6. Perform a scan of the tuples and apply backtracking in $\boldsymbol{S}$ only if the current outer tuple shares the same join matches as the previous one.

We have implemented our approach into the kernel of PostgreSQL, and we have extensively compared our solution against the state of the art techniques both for row- and on column-store DBMSs.

1.3.2 Disjoint Interval Partitioning

The next contribution of this thesis is an efficient partitioning algorithm that allows to efficiently compute operators (namely joins, anti-joins, and aggregations) on temporal relations.

The Disjoint Interval Partitioning ($\mathcal{DIP}$) that we propose divides a relation into the smallest number of partitions with non-overlapping tuples. After applying $\mathcal{DIP}$ to the input relations, efficient sort-merge procedures can be applied between the partitions without computing any backtracking. Our approach is general, simple and systematic: to compute a temporal join, anti-join, or aggregation, we first compute $\mathcal{DIP}$ on the input relations, and then apply a sequence of $\mathcal{DIP}$*Merge*s on the partitions. A $\mathcal{DIP}$*Merge* computes a temporal operator between two $\mathcal{DIP}$ partitions and is efficient, since it is implemented with just one scan of the input partitions.

In this thesis we prove that the number of comparisons done for each tuple in computing a temporal operator using $\mathcal{DIP}$ is linear in the number of partitions. For this reason, our $Create\mathcal{DIP}(\boldsymbol{R})$ algorithm guarantees that the *minimal* number of partitions into which $\boldsymbol{R}$ can be divided is returned. As a result, while sort-merge deteriorates to a quadratic number of comparisons even if just *one* long lived tuple exists in the dataset, with $\mathcal{DIP}$ *all* tuples must overlap to make it quadratic. Furthermore, $Create\mathcal{DIP}(\boldsymbol{R})$ outputs the partitions with their tuples already sorted: before computing a $\mathcal{DIP}$*Merge*, we do not require an additional sorting.

$\mathcal{DIP}$ is the first approach that, at the same time, keeps the number of unproductive comparisons low and does not compute random accesses to the tuples. In this thesis we show that current solutions also leveraging sort-merge (such as the Timeline Index [KMV+13] or the Sweepline [APR+98] algorithm) incur less unproductive comparisons but are slower than $\mathcal{DIP}$ since they suffer from random (disk or memory) accesses: the Timeline Index since it computes one index look up for each matching tuple; Sweepline since after a series of insertions and deletions into the list of active tuples, the elements of the list become randomly scattered in memory [PHD16].

1.3.3 Feedbase.ch

The last contribution of this thesis is identifying the challenges and providing efficient solutions for implementing data warehouse based systems dealing with temporal data. We describe the architecture, interface, functionalities and use cases of Feedbase.ch, a system that uses data from the Swiss Feed Data Warehouse and is used by 3000 Swiss feed mills, research institutions, companies, and private farmers to compose healthy, effective and cheap animal feeding, and to optimize data collection and lab analyzes.

We describe the lessons learned for building efficiently data-driven menus. A *data-driven* menu limits the selections of the user on a given dimension to only the options for which data exists in the fact table satisfying the selections on the previous dimensions. This functionality is essential in data warehouse based systems since the data cube is sparse, i.e., for many combinations of dimensional values data does not exists in the fact table. We combine *partial evaluation* and indexing to avoid to redundantly join the fact table with the dimensions.

We introduce *derived facts* as a new technique to densify the sparse data cube of the Swiss Feed Data Warehouse and compute missing facts. Derived facts are computed in Feedbase.ch for calculating the value of nutrients that have not been measured in the lab on a given feed sample. They are computed applying arithmetic formulas after a series of Nearest Neighbour Joins. We compare three different query evaluation plans for computing derived facts, and show that the one based on group- and selection-enabled NNJ is the most efficient.

We exploit the gain in terms of performances obtained by using *materialized views* to compute distributive aggregates. We compare our runtime against the one of query evaluation plans applied to a denormalized and a normalized fact table. Materialized views are used in the Swiss

Feed Data Warehouse because single dimensions are not selective and computing on-line aggregates when only few dimensions are limited is expensive since many tuples need to be fetched.

1.4 Publications and Organization of the Thesis

This PhD thesis is based on the following research papers:

(i) A GIS-based Data Analysis Platform for Analyzing the Time-Varying Quality of Animal Feed and its Impact on the Environment,

A. Taliun and M. Böhlen and A. Bracher and F. Cafagna. In *Proceedings of the Sixth Biannual Meeting of the International Environmental Modelling and Software Society,*, (iEMSs '12), pages 1447–1454

(ii) Nearest Neighbour Join with Groups and Predicates

Francesco Cafagna, Michael H. Böhlen, and Annelies Bracher. Nearest Neighbour Join with Groups and Predicates. In *Proceedings of the ACM Eighteenth International Workshop on Data Warehousing and OLAP*, DOLAP '15, pages 39–48, ACM, 2015.
DOI: http://dx.doi.org/10.1145/2811222.2811225

(iii) Group- and Selection-Enabled Nearest Neighbour Joins

Francesco Cafagna, and Michael H. Böhlen. Group- and Selection-Enabled Nearest Neighbour Joins. Submitted as research paper.

(iv) Disjoint Interval Partitioning

Francesco Cafagna, and Michael H. Böhlen. Disjoint Interval Partitioning. Submitted as research paper.

(v) Feedbase.ch: a Data Warehouse System for Assessing the Quality of Animal Feed

Francesco Cafagna, Michael H. Böhlen, and Annelies Bracher. Ready for submission.

The rest of this PhD thesis is organized as follows. Chapter 2 is based on paper *(iii)*: it introduces the Group- and Selection-Enabled Nearest Neighbour Join for computing Nearest Neighbour Join queries with robust support for grouping attributes and selection predicates. Chapter 3 is based on paper *(iv)*: it defines the Disjoint Interval Partitioning for computing temporal operators.

Chapter 4 is based on paper *(v)*: it describes Feedbase.ch, i.e., a Data Warehouse Based System used by 3000 Swiss feed mills, research institutions, companies, and private farmers to compose healthy, effective and cheap animal feeding, and to optimize data collection and lab analyzes.

A bibliography for all chapters is given at the end of the thesis.

CHAPTER 2

Group- and Selection-Enabled Nearest Neighbour Joins

Abstract

This Chapter proposes a *group- and selection-enabled* nearest neighbour join (NNJ), $\boldsymbol{R} \ltimes^{T}[\mathbf{G}, \theta] \boldsymbol{S}$, with similarity on T and support for grouping attributes $\mathbf{G}$ and selection predicate θ. Our solution does not suffer from *redundant fetches* and *index false hits*, which are the main performance bottlenecks of current nearest neighbour join techniques.

A *group-enabled* NNJ leverages the grouping attributes $\mathbf{G}$ for the query evaluation. For example, the groups of the query points can be used to limit the fact table to its relevant portions, which guarantees that each block of the fact table is accessed at most once. Solutions that are not group-enabled must process each group independently, and end up fetching (from disk or memory) the blocks of the fact table multiple times. A *selection-enabled* NNJ performs well independent of whether the DBMS optimizer pushes the selection down or evaluates it on the fly. In contrast, index-based solutions suffer from many index false hits or end up in a nested loop.

Our solution does not constrain the physical design, and is efficient for row- as well as column-stores. Current solutions for column-stores use late materialization, which is only efficient if the

fact table is clustered on $\mathbf{G}$. Our evaluation algorithm, $roNNJ$ finds, for each outer tuple r, the inner tuples that satisfy the equality on the group and have the smallest distance to r, with only one scan of both inputs. We experimentally evaluate our solution using the TPC-H benchmark and a data warehouse that manages analyses of animal feeds.

2.1 Introduction

In most real world applications with nearest neighbour joins (NNJs), the nearest neighbours of a tuple $r \in \boldsymbol{R}$ must be determined for a subset of $\boldsymbol{S}$. As an example, consider a data warehouse with a fact table $\boldsymbol{S}$ that stores analyses of animal feeds. If an application asks for the 'Vitamin A' value of 'Soy' on 2014-05-01, we must find the analyses in $\boldsymbol{S}$ with the timestamp closest to 2014-05-01, but only among the tuples that *i*) satisfy *predicate* Nutrient = 'Vitamin A', and *ii)* have the same *group* value (i.e., we want a value for 'Soy'). Towards this end, we propose a group- and selection-enabled NNJ operator, $\boldsymbol{R} \ltimes^{T}[\mathbf{G}, \theta]\, \boldsymbol{S}$, that, for each $r \in \boldsymbol{R}$, returns the tuple with the most similar T value among the tuples in $\boldsymbol{S}$ that have the same group $\mathbf{G}$ and satisfy predicate θ.

In the past, efficient solutions have been developed for computing $\boldsymbol{R} \ltimes^{T} \boldsymbol{S}$, i.e., NNJs without groups and without predicates. These solutions are neither *group-* nor *selection-enabled* and they become inefficient if $\mathbf{G}$ or θ is present. This is a non-trivial problem in NNJ queries since the equality on $\mathbf{G}$ and the evaluation of θ cannot be postponed after the NNJ [SAL+13]. For example, a NNJ might select as nearest neighbour of ('Soy', 2014-05-01) the tuple ('Pea', 2014-05-02). Clearly, this pair is filtered out after evaluating the equality on $\mathbf{G}$ since the groups are different. Thus, no nearest neighbour for ('Soy', 2014-05-01) will be returned, which is incorrect.

To illustrate the performance deterioration of current techniques consider the computation of a NNJ with a B-tree index on $\boldsymbol{S}.T$ [YLK10], grouping attributes $\mathbf{G}$, and predicate θ. The B-tree index allows to quickly find predecessors and successors in time for each $r \in \boldsymbol{R}$. However, if in addition a predicate θ is present the computation suffers from *index false hits* since the predecessors and successors accessed through the B-tree index might not satisfy θ. Especially, if the predicate is selective, most $\boldsymbol{R}$ tuples are compared with each tuple in $\boldsymbol{S}$ and we get $\lim_{sel(\theta)\to 0} \text{Cost}(\textit{B-Tree}) = |\boldsymbol{R}| \times B_{\boldsymbol{S}}$, where $B_{\boldsymbol{S}}$ is the number of blocks of the fact table $\boldsymbol{S}$. As another example, consider the computation of a SortMerge NNJ [SAA10] in combination with the SegmentApply operator [GLJ01] to handle groups. For each group of the query

points $g_i \in \pi_{\mathbf{G}}(\boldsymbol{R})$ the SegmentApply approach fetches the tuples in $\boldsymbol{R}$ and $\boldsymbol{S}$ with group g_i and performs a (group-unaware) NNJ. At the physical level this leads to *redundant fetches* since, for each group, $\boldsymbol{R}$ and $\boldsymbol{S}$ are accessed. If a disk or memory block stores tuples of different groups, this block is fetched multiple times. With $|\pi_{\mathbf{G}}(\boldsymbol{R})|$ groups per block we get: $\lim_{ovlp(\mathbf{G})\to 1} \text{Cost}(\textit{SegApply}) = |\pi_{\mathbf{G}}(\boldsymbol{R})| \times B_{\boldsymbol{S}}$.

Our approach is the first NNJ solution that is *group- and selection-enabled*. It efficiently deals with selective predicates and multiple groups, without constraining the physical organization of the fact table. Our *group-enabled* algorithm is run once independent of the number of groups in the query points. As a result, our query tree copes well with any group-based optimization to reduce the runtime. For example, when the DBMS uses the groups of the query points to limit the tuples of the fact table, our approach still fetches each block at most once. Our *selection-enabled* approach stays robust if the DBMS optimizer either pushes down the evaluation of the predicate before the NNJ (e.g., if θ is selective), or if it evaluates the selection on the fly (e.g., in case θ is always true).

Our approach does not suffer if a given block stores tuples of different groups since the relevant blocks of the fact table are accessed at most once, independent of the number of groups that are stored on a block. Our solution also does not suffer if θ is selective since, in such a case, the tuples that do not satisfy θ are filtered out before the NNJ. Thus, even in a scenario with many groups $\mathbf{G}$ and a very selective predicate θ, we get:

$$\lim_{sel(\theta)\to 0, ovlp(\mathbf{G})\to 1} Cost(roNNJ) = B_{\boldsymbol{S}}$$

The robustness of our solution is independent of the physical design. For example, column-stores perform well only if a primary (or clustered) index on the fact table is available: if the fact table is not clustered by ($\mathbf{G}$, T), redundant fetches are computed on $\boldsymbol{S}.\mathbf{G}$, $\boldsymbol{S}.T$, and on every column involved in θ. Our approach does not require any clustering or index structure, but indexes are leveraged to directly access the tuples of the fact table. The independence of the physical design is a key property of group- and selection-enabled NNJs for two reasons: first, only one clustering can exist; and second, the grouping and similarity attributes are query dependent and change for each query (one NNJ query might compute the similarity on the price, another one on the time, and yet another on the quantity). In our experiments, we show that our approach is up to two orders of magnitude faster than state of the art solutions for computing real world queries on the Swiss Feed Data Warehouse [TBBC12] and on the TCP-H [TPC15] benchmark if no primary index for the grouping and similarity attributes is available.

Our technical contributions are as follows:

- We introduce and define the group- and selection-enabled NNJ operator.
- We introduce an efficient query tree to compute queries with group- and selection-enabled NNJs. Our query tree can be integrated both in row- and column-stores. Independent of the clustering of the fact table, our solution does not suffer from redundant fetches and false hits.
- We provide $roNNJ$, a sort-merge-based algorithm that, for each tuple of the left subtree, finds the tuple in the right subtree that has the same group and the closest value of the similarity attribute, with a single scan of both inputs.
- We describe the seamless integration of NNJ queries with predicates and groups into PostgreSQL.
- We use the Swiss Feed Data Warehouse and the TPC-H benchmark to experimentally evaluate the performance of our approach and compare it with the state of the art techniques implemented on disk, main memory, and column-stores.

The Chapter is organized as follows. Section 2 discusses related work. In Section 3 we present our running example. Section 4 defines the group- and selection-enabled NNJ, and introduces NNJ query trees. In Section 5 we describe our algorithm. Section 6 offers an analytical evaluation of our approach. Section 7 reports the result of an empirical evaluation on the Swiss Feed Data Warehouse and the TPC-H benchmark. Section 8 draws conclusions and points to future work.

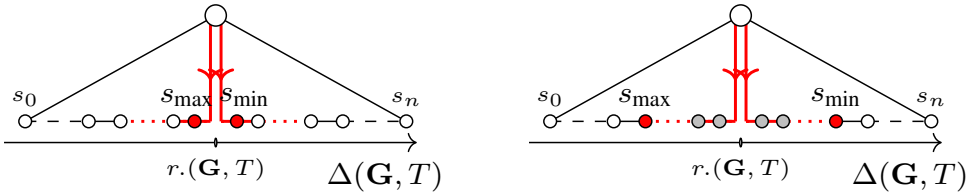
2.2 Related Work

In this section, we introduce the state of the art NNJ solutions and explain the problems they face when dealing with groups and predicates. This Chapter extends the work in Cafagna et al. [CBB15a]. Beyond the contributions of this work we explore the advantages of a query tree with our group-enabled NNJ, and we show that the drawbacks of related approaches are independent of the physical design of the fact table. Towards this goal we implemented and evaluated our solution on column-stores. We show how group- and selection-enabled NNJs can be integrated into the query trees of a column-store (e.g., MonetDB). The experimental

evaluation compares the runtime on column-stores against ours. Due to an increase of the size of our dataset (new analyses have been added to the Swiss Feed Data Warehouse during the last year), the results of the experiments slightly differ compared to the previous paper, especially for the B-Tree since the number of look-ups to compute (two per outer tuple) has increased.

2.2.1 B-Tree

Yao et al. [YLK10] proposed a NNJ implementation with a B-tree index on $(\boldsymbol{S}.\mathbf{G}, \boldsymbol{S}.T)$. This approach performs, for each $r \in \boldsymbol{R}$, two index look-ups in $\boldsymbol{S}$ using $(r.\mathbf{G}, r.T)$ as search key (Figure 2.1.a). One lookup fetches the first tuple to the left (using a MAX subquery), and one lookup fetches the first tuple to the right (using a MIN subqery). The closer of the two tuples is the nearest neighbour of r. Figure 2.1(b) illustrates that this approach suffers from index false



(a) $sel(\theta) = 1$ **(b)** $sel(\theta) < 1$

Figure 2.1: B-Tree implementation where, for $r \in \boldsymbol{R}$, the nearest neighbour is found with a MAX and a MIN query using a B-Tree on $(\boldsymbol{S}.\mathbf{G}, \boldsymbol{S}.T)$. The second figure highlights in grey the false hits of the tuples that do not satisfy predicate θ.

hits if a predicate θ is present. For example the MAX subquery:

`SELECT MAX(T) AS sMax FROM` $\boldsymbol{S}$ `WHERE` $\boldsymbol{S}.\mathbf{G} = r.\mathbf{G}$ `AND` $\boldsymbol{S}.T < r.T$ `AND` θ

no longer guarantees that the maximum is the first tuple that is reached through the index. The index must be scanned to the left until a tuple satisfying θ is found. The higher the selectivity of θ, the higher the number of false hits. This work is index-based and does not investigate index false hits in NNJ query trees.

This drawback is even more prominent in column-stores. Although column-stores avoid to fetch an entire tuple that does not satisfy θ, a single false hit affects each column involved in θ, i.e., for each column a different block must be fetched to evaluate the corresponding predicate. This means that in a column-store each index false hit results in multiple block fetches (and not just in one as for row-stores). We show in our experiments that, while Apache Cassandra [LM10]

adopts this plan if a primary index on $(\mathbf{G}, T)$ exists[1], MonetDB [IGN+12] chooses a different plan: first it fetches the OIDs of the tuples with the same group as r; then T (and each column involved in θ) is scanned and only the OIDs of the entries satisfying $\boldsymbol{S}.T < r.T$ (or predicate θ) are kept; afterwards, the intersection of the OIDs returned from the previous selections is computed; finally, the MAX on the returned tuples is computed. We show in our experiments that this plan, although it avoids the index false hits, is similar to a nested-loop since for a given $r \in \boldsymbol{R}$ it fetches from the fact table all the entries with the same group, and it is therefore expensive.

2.2.2 SegmentApply

Silva et al. [SAA10] proposed a NNJ operator that is not group-enabled, i.e., it computes $\boldsymbol{R} \ltimes^T \boldsymbol{S}$, using SortMerge. This approach sorts $\boldsymbol{R}$ and $\boldsymbol{S}$ by T, and computes the merge step with a single scan of the relations by taking advantage of the order of the tuples. In data warehouses, it is inefficient to sort the entire fact table $\boldsymbol{S}$ and the equality on the groups cannot just be evaluated after the group-unaware NNJ. To manage multiple groups, the SegmentApply operator has been introduced [GLJ01]. It is implemented in DBMSs as lateral subqueries that fetch, for each group $g \in \pi_{\mathbf{G}}(\boldsymbol{R})$, the tuples of $\boldsymbol{S}$ with group g, and runs a SortMerge NNJ:[2]

```
SELECT * FROM R NNJ LATERAL (SELECT * FROM S WHERE G = R.G AND θ) ON T
```

Although this approach avoids that irrelevant portions of the fact table are sorted, it suffers from redundant fetches since it requires a scan of the fact table for each group in $\boldsymbol{R}$. Note that also in the presence of an index on $\boldsymbol{S}.\mathbf{G}$ redundant fetches happen since, if a block stores tuples of m required groups, this block is fetched redundantly m times, once for each group.

In column-stores, for each group $g \in \pi_{\mathbf{G}}(\boldsymbol{R})$, $\boldsymbol{S}.\mathbf{G}$ is accessed and the OIDs of the tuples of group g are returned. The OIDs are joined with the (OID, Value) pairs of each column involved in θ, to select only the tuples that satisfy θ. Finally, their T value is fetched and a group-unaware sort-merge NNJ is run. Without a primary index, i.e., when the fact table is not clustered on $\mathbf{G}$,

[1] In Cassandra the previous query needs to be rewritten as `SELECT` $\mathbf{G}, T$ `FROM` $\boldsymbol{S}$ `WHERE` $\mathbf{G} = r.\mathbf{G}$ `AND` $T \leq r.T$ `AND` θ `ORDER BY` T `LIMIT 1 ALLOW FILTERING` in order to take advantage of the primary index.

[2] Note that the equality on $\mathbf{G}$ cannot be postponed after the NNJ [SAL+13]. Thus, the computation of `SELECT` $*$ `FROM` $\boldsymbol{R}$ `NNJ` $\boldsymbol{S}$ `ON T WHERE` $\boldsymbol{R}.G = \boldsymbol{S}.G$ is not correct, since it first joins each $r \in \boldsymbol{R}$ with its nearest neighbour in $\boldsymbol{S}$ (independently on its group), and then filters out the joined tuples with different groups.

redundant fetches occur since, if m (OID, Value) pairs are stored on the same block and refer to tuples with different groups, then this block is fetched m times. The remaining columns are sequentially scanned and joined to the result after the NNJ for every group is computed, and incur no redundant fetches. If a primary index on $\boldsymbol{S}.\mathbf{G}$ is present, the redundant fetches are almost eliminated, since the (OID, Value) pairs of each column are clustered based on group $\mathbf{G}$. This means that, for each column, the entries of the same group are placed in contiguous blocks. While processing the i-th group, only the first processed block of each column (storing also tuples of the $(i-1)$-th group) will be read redundantly. Note though that the grouping attributes $\mathbf{G}$ are query-dependent and only one primary index can exists: for example, in the Swiss Feed Database, depending on the query, the grouping attributes might specify a biological column (feed type, feed name, stage of maturity, etc.) a geographical column (country, region, postal code, etc.), etc. For completeness we include such a scenario in our experiments, and show that our group- and selection-enabled NNJ stays competitive also in such cases.

2.2.3 Group-Based Optimization Rules

Kimura et al. [KHR+09] introduced the SortedIndexScan to efficiently compute indexed selections on the groups of the query points, i.e., $\sigma_{\mathbf{G}\in\pi_{\mathbf{G}}(\boldsymbol{R})}(\boldsymbol{S})$. This technique traverses the index on $\boldsymbol{S}.\mathbf{G}$ for each needed group (i.e., $\pi_{\mathbf{G}}(\boldsymbol{R})$) and keeps a list of the block IDs that store matching tuples. The block IDs are sorted and deduplicated to avoid fetching multiple times the same block. An equivalent technique has been introduced in the Orca query optimizer [SAR+14]. The Orca query optimizer reduces the number of partitions to fetch, i.e., the relevant blocks, in multi-level partitioned fact tables [AEHS+14]: for each dimension table involved in the join, a *PartitionSelector* scans it and keeps a list of IDs of the partitions of $\boldsymbol{S}$ with join matches. At the end, the intersection of the lists is passed to the *DynamicScanner*, which reads the relevant blocks. We show that our group-enabled query tree can fully leverage such optimizations. As a result, it fetches only the blocks that store tuples with relevant groups (i.e., the groups of $\boldsymbol{R}$), and and it does so once.

2.2.4 Similarity Joins

During the last few years, different similarity join operators have been proposed [BÖ01]: *k-nearest neighbour joins* (k-NNJ) where each outer tuple is joined with the k closest inner tuples, *ϵ joins*

where each outer tuple is joined with all tuples within a given distance range ϵ, *k-distance joins* where the k closest pairs are retrieved, and *join around* where the result is the intersection of an ϵ join and a k-nearest neighbour join. None of those operators integrates groups and predicates.

Works on the evaluation of query trees combining two k-NNJ queries, have been studied by Aly et al. [AAO12]. Note that our implementation assumes $k = 1$ for simplicity but, if many tuples are found at the same minimum distance, returns all of them. Our implementation can be easily adapted to a k-NNJ by using a fixed-size window (of length k) of nearest neighbours.

Partition-based solutions for computing similarity joins have been proposed in the context of Quickjoin and D-Index. Quickjoin [JS08] partitions the data space according to pivot points and creates windows to bind adjacent partitions. Partitions are recursively sub-partitioned until they are small enough to be processed in a memory nested loop. D-Index [DGSZ03] partitions according to a set of mapping functions ρ and uses the D-Index and its extension called eD-Index [DGZ03] to access a small portion of data within which the closest pairs are searched. Quickjoin focuses on ϵ joins and requires a rebuild of the index for each different ϵ value; D-Index has been introduced for computing self-joins. Both approaches do not consider groups and predicates, and have not been integrated into a DBMS.

2.3 Running Example

As a running example, we use the Swiss Feed Data Warehouse [TBBC12], i.e., a data warehouse that stores lab analysis of animal feeds, using a fact table with a vertical design where each value of the analysis is stored in a different row [KR02; LÖ09]. Figure 2.2 shows selected tuples of fact table $\boldsymbol{S}$. Animal feeds G, such as 'Soy', 'Pea', or 'Hay', are sampled in the field at an altitude A and analyzed at time T in a lab where the content M of various nutrients N is measured with reliability R. For instance, tuple s_0 records that for feed 'Soy', grown at an altitude of 1030 meters, the nutrient content of 'CP' (Crude Protein) at time 2014-06-15 is 1.40 with reliability 0.9. Since lab analyses are expensive and more than 600 nutrients exist, not all nutrients N are measured on a daily basis (e.g., no 'CP' value has been measured for 'Soy' on 2014-06-19).

Relation $\boldsymbol{R}$ in Figure 2.2 stores the outer tuples for which the nearest neighbours in $\boldsymbol{S}$ must be retrieved: attributes G and T correspond to the feed and day for which a measurement is needed. In the web application of the Swiss Feed Data Warehouse (http://www.feedbase.ch), the users (farmers, domain experts, etc.) use the result of the NNJs to compute graphical

$\boldsymbol{R}$

	G	T
r_0	Soy	2014-06-15
r_1	Soy	2014-06-21
r_2	Pea	2014-06-20

$\boldsymbol{S}$

	G	T	A	R	N	M
s_0	Soy	2014-06-15	1030	0.9	CP	1.40
s_1	Soy	2014-06-20	1000	1.0	CP	1.08
s_2	Soy	2014-06-21	1020	0.5	CP	0.93
s_3	Soy	2014-06-27	1110	0.9	CP	1.23
s_4	Pea	2014-06-19	1000	0.8	CP	4.20
s_5	Pea	2014-06-20	1000	0.3	CP	4.10
s_6	Pea	2014-06-21	1100	0.9	CP	4.03
s_7	Hay	2014-06-19	1000	0.8	OM	0.32

Figure 2.2: Outer Relation $\boldsymbol{R}$; Fact Table $\boldsymbol{S}$ with Lab Analyses of the Nutrients of Feeds.

interpolations that represent the evolution of a given nutrient in different feeds to pick the feed that best suits the desired characteristics (e.g., the *cereal* that has the most stable protein content in the animal feeding process). Attribute G in $\boldsymbol{R}$ typically covers up to 5% of the feeds stored in the fact table (e.g., all cereals), while attribute T represents the days for which the nutritive values must be computed. Note that it is not possible to precompute the join off-line since the result depends on predicate θ, which is defined over a combination of attributes that change for each query, e.g., $\theta \equiv (N =$ 'CP' $\wedge\, R > 0.7)$.

Table 2.1 summarizes the notation we use in this Chapter.

Table 2.1: Notation.

Symbol	Meaning	Example
$\mathbf{G}$	grouping attribute set	G
T	similarity attribute	T
θ	predicate on $\mathbf{S}$	$R > 0.7$
$sel(\theta)$	predicate selectivity	$\lvert\sigma_{R>0.7}(\boldsymbol{S})\rvert \,/\, \lvert\boldsymbol{S}\rvert$
$sel(\mathbf{G})$	group selectivity	$\lvert\sigma_{\mathbf{G}\in\pi_{\mathbf{G}}(\boldsymbol{R})}(\boldsymbol{S})\rvert \,/\, \lvert\boldsymbol{S}\rvert$

2.4 A Group- and Selection-Enabled Nearest Neighbour Join

We assume a multidimensional schema $\mathbf{S} = [\mathbf{G}, T, \mathbf{A}]$ with attributes $G_1, ..., G_m, T, V_1, ..., V_n$. A tuple s over schema $\mathbf{S}$ is a *fact*. The *fact table* $\boldsymbol{S}$ is a relation over schema $\mathbf{S}$. We write $|\boldsymbol{S}|$

for the number of tuples in $\boldsymbol{S}$. For a tuple $s \in \boldsymbol{S}$ and an attribute V_i, $s.V_i$ denotes the value of attribute V_i. We assume a totally ordered similarity attribute T. The concatenation operator $r \circ s$ appends to r the attributes of s.

Definition 1. Assume relation $\boldsymbol{R}$ with schema $\mathbf{R} \supseteq [\mathbf{G}, T]$ and fact table $\boldsymbol{S}$ with schema $\mathbf{S} = [\mathbf{G}, T, \mathbf{A}]$. Let θ be a predicate on $\mathbf{S}$. The *Group- and Selection-Enabled Nearest Neighbour Join*, $\boldsymbol{R} \ltimes^T [\mathbf{G}, \theta]\ \boldsymbol{S}$, returns, for a given tuple $r \in \boldsymbol{R}$, the tuples $s \in \sigma_\theta(\boldsymbol{S})$ with the same group $\mathbf{G}$ that have the closest T value:

$$\boldsymbol{R} \ltimes^T [\mathbf{G}, \theta]\ \boldsymbol{S} = \big\{ r.\mathbf{R} \circ s.\mathbf{A} \mid r \in \boldsymbol{R} \wedge s \in \sigma_\theta(\boldsymbol{S}) \ \wedge\ r.\mathbf{G} = s.\mathbf{G} \wedge \nexists t \in \sigma_\theta(\boldsymbol{S}) \big(r.\mathbf{G} = t.\mathbf{G} \wedge |r.T - t.T| < |r.T - s.T| \big) \big\}$$

Example 5. Consider in Figure 2.2 relation $\boldsymbol{R}$ with schema $\mathbf{R} = [G, T]$, fact table $\boldsymbol{S}$ with schema $\mathbf{S} = [G, T, A, R, N, M]$, and predicate $\theta \equiv (N =$ 'CP' $\wedge\ R > 0.7)$. We apply Definition 1 with $\mathbf{G} = G$ and $\mathbf{A} = A, R, N, C$ to compute $\boldsymbol{Z} = \boldsymbol{R} \ltimes^T [G, N =$ 'CP' $\wedge\ R > 0.7]\ \boldsymbol{S}$. Thus, we join each outer tuple in $\boldsymbol{R}$ with the temporally closest 'CP' measure having reliability greater than 0.7. Result relation $\boldsymbol{Z}$ is shown in Figure 2.3. For example, the nearest neighbour of tuple r_0 is fact s_0 since it is the closest. For tuple r_1, s_1 is its nearest neighbour. Even if temporally closer, s_2 has not been chosen as nearest neighbour since it does not satisfy $R > 0.7$. For tuple r_2, two facts (s_4, s_6) exist at the same minimum distance, and therefore two join matches are returned.

$\mathbf{Z} = \boldsymbol{R} \ltimes^T [G, N =$ 'CP' $\wedge\ R > 0.7]\ \boldsymbol{S}$

	G	T	A	R	N	C
$r_0 s_0$	Soy	2014-06-15	1030	0.9	CP	1.40
$r_1 s_1$	Soy	2014-06-21	1000	1.0	CP	1.08
$r_2 s_4$	Pea	2014-06-20	1000	0.8	CP	4.20
$r_2 s_6$	Pea	2014-06-20	1100	0.9	CP	4.03

Figure 2.3: NNJ Result $\mathbf{Z}$.

The NNJ in our example uses the feed name as a grouping attribute and the time as similarity attribute. For the similarity attribute T any attribute whose domain is totally ordered can be used (e.g., similarity on the price, the quantity, the time, etc.)

2.4.1 The $roNNJ$ Query Tree

In DBMSs joins are group- and selection-enabled, since they support grouping attributes $\mathbf{G}$ and selections θ. For example, for a join $\boldsymbol{R} \bowtie_{\boldsymbol{R}.\mathbf{G}=\boldsymbol{S}.\mathbf{G}\wedge\theta} \boldsymbol{S}$, each joined pair must satisfy the equality on $\mathbf{G}$ and selection θ. A group- and selection-enabled join has two important implications: first, the join is computed once, independent of the number of groups in the data; second, the DBMS can take advantage of any optimizations on $\mathbf{G}$ and θ to improve performance. For example, if θ is selective, its evaluation can be pushed down before the join to reduce the number of tuples to process. Ours is the first NNJ solution that remains efficient when combined with other DBMS optimizations. Current solutions are not robust and easily degenerate with many index false hits and redundant fetches.

The query tree in Figure 2.4(a) illustrates the base case of our approach. The top node $\ltimes^{T}[\mathbf{G},\theta]^{\textbf{Merge}}$ of the tree is *group-* and *selection-enabled*, and represents our $roNNJ$ algorithm (cf. Section 2.5). For each tuple of the left subtree the $\ltimes^{T}[\mathbf{G},\theta]^{\textbf{Merge}}$ node finds in the right subtree the nearest neighbours according to T among the tuples that satisfy the equality on $\mathbf{G}$ and condition θ. The group-enabled NNJ node, $\ltimes^{T}[\mathbf{G},\theta]^{\textbf{Merge}}$, computes its left and right subtrees only once, independent of the number of groups, and can fully leverage other optimizations.

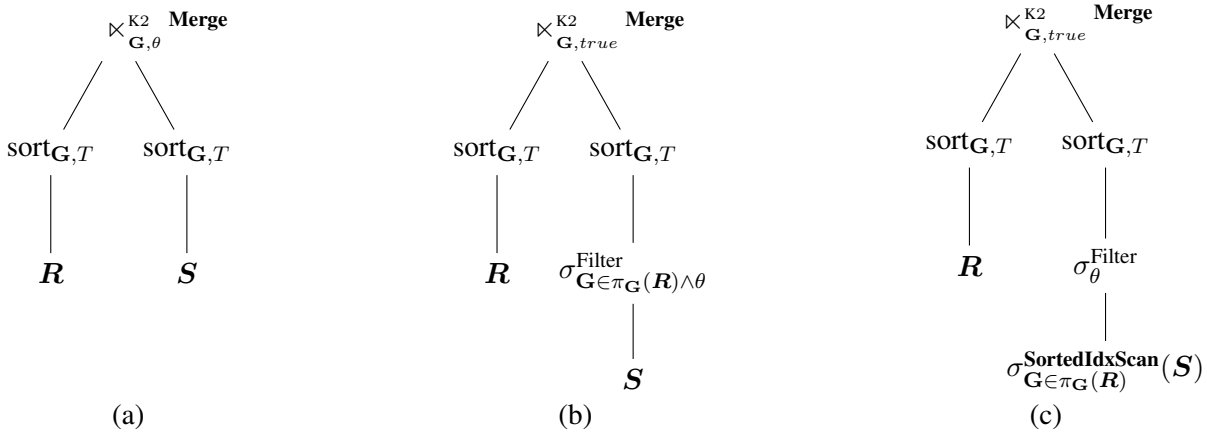


Figure 2.4: A Group- and Selection-Enabled Nearest Neighbour Join Tree accesses the Fact Table $\boldsymbol{S}$ only once. It can take advantage of all optimizations offered by the DBMS.

For example, the tree in Figure 2.4(b), prior to computing the NNJ, applies *selection pushdown* [Ull90]. The right subtree makes an additional scan on $\boldsymbol{R}$ (for reading the groups stored in it) and selects from $\boldsymbol{S}$ only the tuples with the groups of $\boldsymbol{R}$. Similarly, it pushes down the evaluation of θ. The selection node passes to the $sort_{\mathbf{G},T}$ node only the portion of the fact table that contributes

to the NNJ result, i.e., the tuples that have the same groups as the tuples in $\boldsymbol{R}$ and satisfy θ. This avoids the sort of the entire fact table.

In Figure 2.4(c) we show that, if an index on $\boldsymbol{S}.\mathbf{G}$ is present, the selection on $\mathbf{G}$ allows the DBMS to fetch only the blocks of the fact table that are relevant to the join. The indexed selection $\sigma_{\mathbf{G}\in\pi_{\mathbf{G}}(\boldsymbol{R})}(\boldsymbol{S})$ is implemented using a *SortedIndexScan*[3] [KHR+09]. It does not fetch blocks that do not store a tuple with the groups of $\boldsymbol{R}$ since they will not contribute to the result. After the relevant blocks have been fetched, only the tuples with a matching group are kept. The tree in Figure 2.4(c) also shows how to efficiently combine the push-down of selections on θ with a SortedIndexScan on $\mathbf{G}$. In such a case, condition θ must be evaluated *before* the join, but *after* the selection on the group $\mathbf{G}$. This allows to compute the selection on $\mathbf{G}$ using the index, i.e., without a full scan of the fact table $\boldsymbol{S}$; and it allows to compute σ_θ on the fly (almost) for free when the relevant portion of the fact table is retrieved.

Current NNJ solutions $\ltimes^T$ are not group- and selection-enabled, which limits the scope of the query optimizer. A group-unaware NNJ will join tuples of different groups, which is wrong. Solutions based on the SegmentApply [GLJ01] operator process groups individually, and fetch for each group of $\boldsymbol{R}$, the portion of the fact table storing tuples of that group. If a block of $\boldsymbol{S}$ stores tuples with different groups the block is fetched multiple times. This is inefficient for big fact tables. Solutions based on a B-tree [YLK10], suffer from index false hits if θ is evaluated on the fly.

2.4.2 The $roNNJ$ Query Tree in Column-Stores

This subsection shows that group- and selection-enabled NNJs can also be efficiently integrated into column-stores. The query tree[4] combines both *early materialization* [HLAM06] and *late materialization* [AMDM07]: the former, since for computing $sort_{\mathbf{G},T}$ before the NNJ, columns[5] $\mathbf{G}$ and T need to be combined; the latter, since the rest of the columns of the fact table are fetched after the NNJ has been computed. In general, column-stores try to avoid early materialization to keep the data small. For example, in a NNJ query, they select the oid of the entries with group g,

[3] A sorted index scan, when all values of an IN-subquery are known up-front (i.e., $\pi_{\mathbf{G}}(\boldsymbol{R})$), performs for each value an index look-up on $\boldsymbol{S}$ and collects a list of the IDs of the (relevant) blocks storing matching tuples. It then sorts and deduplicates the list, and fetches each of the relevant blocks once in sorted order.

[4] We use MonetDB 11.21.5 [IGN+12] as a reference point.

[5] When we draw a node operating on an attribute set (e.g., $\mathbf{G} = G_1, \ldots, G_g$), the node is intended to be replicated for each attribute of the set.

fetch their T value and compute the NNJ just using T, for each $g \in \Pi_{\mathbf{G}}(\boldsymbol{R})$. The other attributes are only fetched at the very end to construct the result tuples. Such an approach incurs redundant fetches when the fact table is not clustered on $\mathbf{G}$ (remember that $\mathbf{G}$ is not fixed and changes for different NNJ queries) and makes NNJ queries slow.

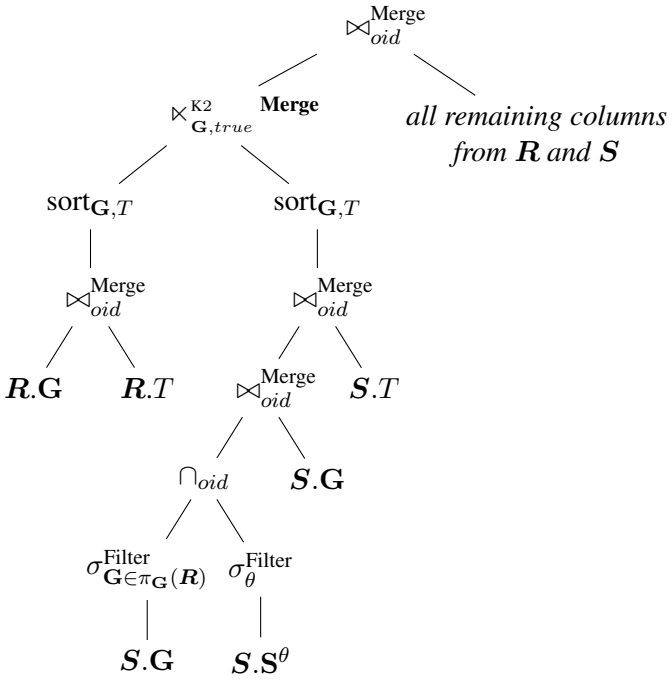


Figure 2.5: Each node in a Robust Nearest Neighbour Join Tree for column store DBMSs returns only oids (e.g., the selections σ) so that the parent node can take advantage of the order with which the (oid, $value$) pairs are physically stored. The actual column values are returned by a join $\bowtie$ node to the parent node only when the are needed.

Figure 2.5 illustrates the query tree for our running example in MonetDB. In the left subtree of Figure 2.5, first $\boldsymbol{R}.\mathbf{G}$ is joined to $\boldsymbol{R}.T$. This is not expensive because the columns of $\mathbf{G}$ and T are stored with the same oid order, and the join can be performed with a scan (i.e., a merge) of two columns. Afterwards, the join result is sorted by the values of ($\mathbf{G}$,T). In the fact table $\boldsymbol{S}$, $\boldsymbol{S}.\mathbf{G}$ is accessed and the oids of the tuples with the groups of $\boldsymbol{R}$ are returned. Column-stores implement such a condition as an *invisible join* [AMH08], which rewrites the semijoin between $\boldsymbol{R}$ and $\boldsymbol{S}$, as a selection predicate. It is implemented by scanning $\boldsymbol{S}.\mathbf{G}$ and comparing it with the entries of $\pi_{\mathbf{G}}(\boldsymbol{R})$, which have been stored in a hash-table. Note that if the NNJ was not group-enabled, such an optimization would be useless since, in order to ensure the correctness of the result, only one group at a time may be passed to the NNJ node.

Subsequently, each column used in the θ condition (we refer to them as S^{θ}) is accessed, and the oids of the tuples satisfying θ are returned. They are intersected with the oids of the tuples of the selected groups. In column-stores the order of the oids of a selection is preserved. Thus, the lists of oids of the two selections come in the same order, and their intersection can be performed by the parent node $\cap_{oid}$ with only a scan of the two lists. Afterwards the values of $\boldsymbol{S}.\mathbf{G}$ and $\boldsymbol{S}.T$ are fetched and joined to the previous oids (again, the join is performed with a scan). Finally, $roNNJ$ is applied as a SortMerge procedure, and the remaining columns are concatenated to the result through a join. Thus, late materialization is used for all attributes apart from $\mathbf{G}$ and T.

Summarizing, our approach is independent of the physical layout of the relations: it avoids index false hits and redundant fetches also in column-stores. Essentially, each column is read only once.

2.5 The Robust NNJ Algorithm

This section describes the $roNNJ$ algorithm, i.e., the implementation of the top node of our query tree into the kernel of PostgreSQL. We give the details of the extension, and present an efficient algorithm that: 1) computes the join with a single access of the input relations, i.e., without redundant fetches; and 2) does not suffer from false hits, i.e., each tuple that is not a nearest neighbour is not read more than once.

2.5.1 Algorithm Properties

Sort merge has been proposed as a method for computing equijoins [BE77] and as a method for computing nearest neighbour joins [SAA10]. We describe an implementation that efficiently combines those two approaches (equijoin on the group, and nearest neighbour join on the similarity attribute), and that does not do unnecessary backtracking. As reference point we use the tree in Figure 2.4(b). This tree applies selection push-down as a DBMS optimization on $\mathbf{G}$ and θ.

The algorithm takes the two input relations $\boldsymbol{R}$ and $\boldsymbol{S}^{\mathbf{G},\theta} \equiv \sigma_{\mathbf{G}\in\pi_{\mathbf{G}}(\boldsymbol{R})\wedge\theta}(\boldsymbol{S})$, and sorts them by $(\mathbf{G}, T)$. The sorting by $\mathbf{G}$ guarantees that the tuples are grouped according to their group value. Thus, as illustrated in Figure 2.6, all tuples of feed 'Hay' (as well as all 'Pea' and 'Soy') are adjacent. This means that, while processing an outer tuple $r \in \boldsymbol{R}$ with group 'Pea', no

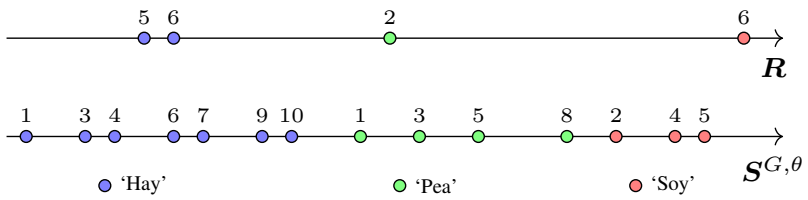


Figure 2.6: Relations $\boldsymbol{R}$ and $\boldsymbol{S}^{G,N=\text{'CP'}} \equiv \sigma_{G \in \pi_G(\boldsymbol{R}) \wedge N=\text{'CP'}}(\boldsymbol{S})$ sorted by group G (represented with the colour) and similarity attribute T (represented with the number). No tuple may be available for a given timestamp: for example, a gap between ('Pea',1) and ('Pea', 3) shows that no 'Pea' tuple is available for $T = 2$.

backtracking in $\boldsymbol{S}$ to the tuples of group 'Hay' is needed since the nearest neighbours must satisfy the equality on the groups. The sorting by T makes sure that tuples in $\boldsymbol{S}^{\mathbf{G},\theta}$ that have been previously read but that were not nearest neighbours do not have to be read again. For example, for $r = (\text{'Hay'}, 5)$, the inner relation $\boldsymbol{S}^{\mathbf{G},\theta}$ is scanned until its nearest neighbours $(\text{'Hay'}, 4)$ and $(\text{'Hay'}, 6)$ are reached. For the following $\boldsymbol{R}$ tuple, no tuple before $(\text{'Hay'}, 4)$ has to be considered from $\boldsymbol{S}^{\mathbf{G},\theta}$ since it will have a higher distance than $(\text{'Hay'}, 4)$ itself. We now prove that our algorithm does not fetch more than once any tuple that is not nearest neighbour.

Lemma 1. *(No Unnecessary Backtracking) In computing $\boldsymbol{R} \ltimes^{T}[\mathbf{G}, \theta]\, \boldsymbol{S}^{\mathbf{G},\theta}$ using a sort-merge computation, every tuple that is not a nearest neighbour is read at most once.*

Proof. In a sort-merge computation, during the merge, tuples might be read more than once only if *backtracking* (i.e., going back to a previously scanned row) is applied. We now show that backtracking, when applied, fetches only the nearest neighbours. Let $\{r_{i-1}, r_i\} \subseteq \boldsymbol{R}$, r_i be the current tuple for which the nearest neighbours have to be found, and $d(r_i, s) = |r_i.T - s.T|$ be the absolute distance between tuples r_i and s. Since the input relations are sorted by $\mathbf{G}, T$ three cases are possible:

1. $r_{i-1}.\mathbf{G} < r_i.\mathbf{G}$: straightforward. Since the nearest neighbours of r_i must have its same group, then they will, for sure, be in $\boldsymbol{S}$ after all tuples of group $r_{i-1}.\mathbf{G}$. No backtracking needs to be applied.

2. $r_{i-1}.\mathbf{G} = r_i.\mathbf{G} \wedge r_{i-1}.T < r_i.T$. Let s_j be the first nearest neighbour of r_{i-1}: since r_i succeeds r_{i-1}, then $d(r_i, s_j) = d(r_{i-1}, s_j) + d(r_{i-1}, r_i)$. For any $k < j$ such that s_k shares the same group of r_i (if s_k has a different group, case 1 re-applies), $r_i.\mathbf{G} = s_k.\mathbf{G} \Rightarrow d(r_i, s_k) = d(s_k, s_j) + d(s_j, r_{i-1}) + d(r_{i-1}, r_i)$. Since $s_k.T < s_j.T$, then $d(s_k, s_j) > 0$:

any tuple s_k preceding the first nearest neighbour (s_j) of r_{i-1} will have a bigger distance to r_i than s_j itself, i.e., $d(r_i, s_k) > d(r_i, s_j)$. No backtracking needs to be applied.

3. $r_{i-1}.\mathbf{G} = r_i.\mathbf{G} \wedge r_{i-1}.T = r_i.T$. Tuples r_{i-1} and r_i share the same nearest neighbours, and backtracking to the nearest neighbours of r_{i-1} *has* to be applied. Let s_j and s_k be, respectively, the first and the last nearest neighbours of r_{i-1}. In order for Lemma 1 to hold, we must make sure that between s_j and s_k no tuple exists that is not a nearest neighbour for r_i. Being s_j and s_k nearest neighbours, then they must have the same (minimum) distance from r_i, i.e., $d(r_i, s_j) = d(r_i, s_k)$. However, this is true only if $d(r_i, s_j) = d(r_i, s_k) \Leftrightarrow s_j.T = (r_i.T - \epsilon) \wedge s_k.T = (r_i.T + \epsilon)$, with $\epsilon \geq 0$. A tuple $s \in \boldsymbol{S}$ such that $r_i.T - \epsilon < s.T < r_i.T + \epsilon$ cannot exists, otherwise it would be nearest neighbour itself, instead of s_j and s_k.

□

The above Lemma proves that our approach computes a NNJ with only one scan of the input relations. The only tuples that our algorithm rescans through backtracking are the nearest neighbours (if two outer tuples r_i and r_{i+1} share the same result tuples). For example, for tuple r_2 of our main example, s_4 and s_6 are the nearest neighbours. Removing s_5 before the sorting, ensures that s_4 and s_6 will be adjacent elements in $\boldsymbol{S}$. Backtracking would only be needed if a tuple r_3 existed with s_4 and s_6 as its nearest neighbours.

2.5.2 The *roNNJ* Algorithm

We now describe the *roNNJ* implementation of the NNJ operator. The SortMerge *roNNJ* has been implemented as a set of states (cf. Figure 2.7), similar to the traditional sort merge join in PostgreSQL.

When the actual node to compute is a NNJ, the executor of the DBMS calls a procedure **Exec_roNNJ**(*NNJObj*), where *NNJObj* is an object shared by all states, storing, among others: ***OuterPlan** (a reference to the tuples coming as input from the left subtree of Figure 2.4), ***InnerPlan** (a reference to the tuples coming as input from the right subtree of Figure 2.4), $\mathbf{G}$ (the position of the grouping attributes in the schema of $\boldsymbol{R}$ and $\boldsymbol{S}$), T (the position of the similarity attribute), r (the current outer tuple for which the nearest neighbours have to be found), s_c (the current inner tuple), s_n (the next inner tuple), *nextState* (the state to be executed in the next iteration of the

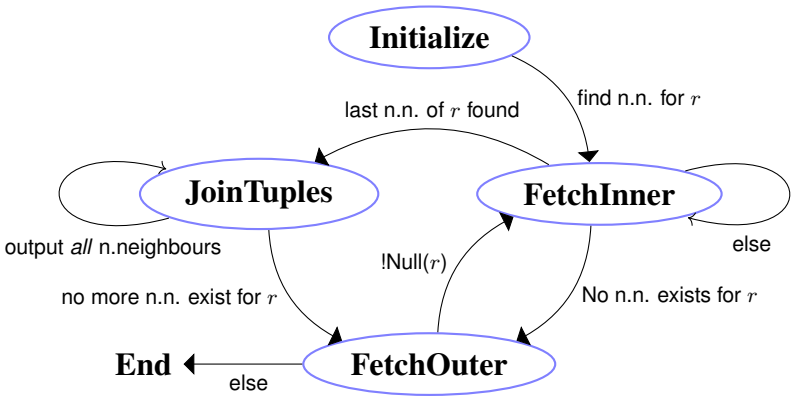


Figure 2.7: State diagram of Exec_roNNJ

algorithm). The reader can see that, opposite to an equijoin node (where only the current tuples are stored), for a NNJ both the current and the next $\boldsymbol{S}$ tuples are needed. This is so since, after the sorting, we can tell if s_c is the nearest neighbour of r only after comparing its distance with the one of s_n.

The procedure is shown in Algorithm 1: it consists of a loop in which, at each iteration, one state is executed and the object *NNJObj* is modified. *NNJObj* is initially set to $(\boldsymbol{R}, \boldsymbol{S}^{\mathbf{G},\theta}, \mathbf{G}, T, null, null, null, 1)$, where $\boldsymbol{S}^{\mathbf{G},\theta} \equiv \sigma_{\mathbf{G}\in\pi_{\mathbf{G}}(\boldsymbol{R})\wedge\theta}(\boldsymbol{S})$, i.e., the right subtree of Figure 2.4. In each state tuples are fetched, joined, etc., and the next state to be executed is set. For shortness, in each state, we omit the name of the object *NNJObj* in front of each variable, e.g., we write r instead of *NNJObj*.r.

The following subsections describe the four states of $roNNJ$. For a given $r \in \boldsymbol{R}$, $roNNJ$ returns *all* its nearest neighbours, i.e., all tuples minimizing their distance to r, independent of their number.[6]

Initialize

In this state, we start the scan of the two relations. We initialize r with the first outer tuple, and s_c and s_n with the first inner tuple. The next state to be computed is FetchInner.

[6] For k-NNJ queries, joining r with the k closest tuples, a similar implementation with a window of k $\boldsymbol{S}$ tuples from s_c to s_n can be used. The window is moved until s_n is more far than s_c to r or has a different group.

Algorithm 1: Exec_roNNJ($\boldsymbol{R}$, $\boldsymbol{S}^{\mathbf{G},\theta}$, $\mathbf{G}, T, null, null, null, 1$)

```
Input: NNJObj.{*OuterPlan,                          // R sorted by G,T
               *InnerPlan,                          // S sorted by G,T
               G,                                   // grouping attribute
               T,                                   // similarity attribute
               r,                                   // current R tuple
               s_c,                                 // current S tuple
               s_n,                                 // next S tuple
               nextState}                           // state to perform next
1 begin
2   while true do
3     switch nextState do
4       case 1: Initialize(NNJObj)
5       case 2: FetchInner(NNJObj)
6       case 3: JoinTuples(NNJObj)
7       case 4: FetchOuter(NNJObj)
```

State 1: Initialize(*NNJObj*)

```
1 r ← fetchRow(OuterPlan)
2 s_c ← fetchRow(InnerPlan)
3 markPosition(S)
4 s_n ← s_c
5 nextState = 2                                     // Go to FetchInner
```

FetchInner

In this state we fetch the next inner tuple. First, in lines 1-3 we check if an inner tuple with the same group as the actual outer tuple exists at all: if not, no join match exists for r, and we go to FetchOuter. In lines 4-8 we fetch a new inner tuple: if r is closer to s_n than to s_c, then s_n *might* be its (first) nearest neighbour. Therefore we mark its position. In lines 9 - 13, as soon as the next tuple s_n has a higher distance to r or belongs to a different group, we are sure that the previously marked tuple is its first nearest neighbour (the tuple marked in line 6) and no more nearest neighbours exist. We, therefore, restore s_c to the first nearest neighbour and we go to the state JoinTuples.

State 2: FetchInner(*NNJObj*)

```
1  if r.G < s_c.G then
2  |   nextState = 4                                   // No NN exists for r
3  |   break
4  if !Null(s_n) then
5  |   if r.G = s_n.G ∧ (d(r, s_c) > d(r, s_n) ∨ r.G ≠ s_c.G) then
6  |   |   markPosition(InnerPlan)                     // May be first NN of r
7  |   s_c ← s_n
8  |   s_n ← fetchRow(InnerPlan)
9  if r.G = s_c.G ∧ (Null(s_n) ∨ d(r, s_c) < d(r, s_n) ∨ r.G ≠ s_n.G) then
10 |   s_c ← restorePosition(InnerPlan)                // Fetch first NN of r
11 |   s_n ← fetchRow(InnerPlan)
12 |   nextState = 3                                   // GoTo JoinTuples
13 |   break
14 nextState = 2                                      // Last NN of r not yet reched
```

Join Tuples

In this state we join r with all its nearest neighbours (for a given outer tuple, multiple nearest neighbours might exist). We scan the inner relation from the position marked in the state FetchInner, and we join r with s_c. If s_n has a bigger distance to r than s_c, then s_c was the last nearest neighbour of r and we go to FetchOuter. If s_n does not have a bigger distance than s_c, then s_n is also a nearest neighbour for r, and we do not change state.

State 3: JoinTuples(*NNJObj*)

```
1  Output(r ∘ s_c)                                    // Output Result Tuple
2  if Null(s_n) ∨ d(r, s_n) > d(r, s_c) ∨ r.G ≠ s_n.G then
3  |   nextState = 4                                  // All NNs of r are found
4  else
5  |   s_c ← s_n
6  |   s_n ← fetchRow(InnerPlan)                      // Still NNs to fetch
```

Fetch Outer

In this state, we first fetch a new outer tuple. Then, in case r shares the same join matches of the previous outer tuple (lines 3-5), we go back in the inner relation to its first nearest neighbour: this is the only backtracking that our algorithm performs. We then jump to state FetchInner. In case no more outer tuples exist, the algorithm ends.

State 4: FetchOuter(*NNJObj*)

```
1 r ← fetchRow(OuterPlan)
2 if !Null(r) then
3 |   if d(r, s_c) ≤ d(r, s_n) ∨ Null(s_n) then
4 |   |   restorePosition(InnerPlan)
5 |   |   s_n ← fetchRow(InnerPlan)
6 |   nextState = 2                                  // Search the NNs of r
7 |   break
8 End                                               // No more tuples to process
```

2.5.3 SQL Syntax Extension

The NNJ operator can be used similar to the standard join operators, except that since our operator is group-embedded, a second argument (i.e., the group $\mathbf{G}$) must be specified. The concrete SQL syntax for $\boldsymbol{R} \ltimes^{\text{K2}}_{\mathbf{G},\theta} \boldsymbol{S}$ is:

```
SELECT *
FROM R NNJ S ON T USING G
WHERE θ
```

The keyword `NNJ` specifies the join operator, `ON` specifies the similarity attribute T, and `USING` the grouping attribute $\mathbf{G}$. Condition θ can be specified along with any other conditions in the `WHERE` clause of the SQL query.

As an example consider an SQL query from the Swiss Feed Data Warehouse that computes the *derived nutrient* 'GE' (Gross Energy). The Gross Energy value of the feed samples in $\boldsymbol{R}$ is calculated as follows: i) one NNJ to retrieve the closest measurement of nutrient 'CP' (Crude Protein) in $\boldsymbol{S}$, i.e., $N =$ 'CP'; ii) one NNJ to retrieve the closest measurement of nutrient 'OM' (Organic Matter) in $\boldsymbol{S}$, i.e., $N =$ 'OM'; and iii) evaluation of the formula $0.8 * CP + 2 * OM$ on the join result.

```
SELECT G, T, 'GE', 0.8 * CP + 2 * OM
FROM (SELECT Z1.*, M AS OM
      FROM (SELECT R.*, M AS CP
            FROM R NNJ S ON T USING G
            WHERE N = 'CP') AS Z1 NNJ S ON T USING G
      WHERE N = 'OM') AS Z2
```

Note that no materialization of intermediate join results is needed. As soon as an output tuple of $\boldsymbol{Z}_1$ is produced, it can be pipelined to compute $\boldsymbol{Z}_2$.

2.6 Complexity of Query Tree

This section computes the number of operations on the fact table $\boldsymbol{S}$ in terms of disk I/Os, memory I/Os and CPU operations. We prove that, independent of the number of groups and of the selectivity of θ, our approach is upper bounded by a complexity of $n \log n$. For simplicity, we exclude the cardinality of the result from our analysis since only in very special cases (e.g., when all the tuples have the same value for group and similarity attribute) it's bigger than $n \log n$.

Lemma 2. *(Disk) Let $\boldsymbol{R}$ be a relation with schema $\mathbf{R} \supseteq [\mathbf{G}, T]$, $\boldsymbol{S}$ be a fact table with schema $\mathbf{S} = [\mathbf{G}, T, \mathbf{A}]$, and θ be a predicate on $\mathbf{S}$. The number of disk I/Os on the fact table $\boldsymbol{S}$ of a Data Warehouse, for computing $\boldsymbol{R} \ltimes^{T} [\mathbf{G}, \theta]\, \boldsymbol{S}$, is lower bounded by a linear complexity and is upper bounded by a linearithmic complexity.*

Proof. Consider the number of Disk I/Os for computing each operation in Figure 2.4(b).

$$
\begin{aligned}
&Disk(\text{read}(\boldsymbol{R})) = B_{\boldsymbol{R}} \\
&Disk(\text{read}(\boldsymbol{S})) = B_{\boldsymbol{S}} \\
&Disk(\text{sort}(\boldsymbol{R})) = B_{\boldsymbol{R}} \log_M B_{\boldsymbol{R}} \\
&Disk(\text{sort}(\sigma(\boldsymbol{S})) = \big(sel(\theta) * sel(\mathbf{G}) * B_{\boldsymbol{S}}\big) \log_M \big(sel(\theta) * sel(\mathbf{G}) * B_{\boldsymbol{S}}\big) \\
&Disk(\text{merge}(\boldsymbol{R}, \sigma(\boldsymbol{S})) = B_{\boldsymbol{R}} + sel(\theta) * sel(\mathbf{G}) * B_{\boldsymbol{S}}.
\end{aligned}
$$

$M = max(mem\_size, B)$ is the memory available for a relation with B blocks, $sel(\theta) = \frac{|\sigma_\theta(\boldsymbol{S})|}{|\boldsymbol{S}|}$ is the *predicate selectivity*, with $0 \leq sel(\theta) \leq 1$, and $sel(\mathbf{G}) = \frac{|\sigma_{\mathbf{G} \in \pi_{\mathbf{G}}(\boldsymbol{R})}(\boldsymbol{S})|}{|\boldsymbol{S}|}$ is the *group selectivity*, with $0 \leq sel(\mathbf{G}) \leq 1$.

The selection $\sigma(\boldsymbol{S})$ is evaluated on the fly while reading the $B_{\boldsymbol{S}}$ blocks of $\boldsymbol{S}$ and incurs no additional I/Os. This selection returns $sel(\theta) * sel(\mathbf{G}) * B_{\boldsymbol{S}}$ blocks: $sel(\theta) * B_{\boldsymbol{S}}$ are needed for the tuples satisfying θ, and $sel(\mathbf{G}) * B_{\boldsymbol{S}}$ blocks are needed for the tuples with the groups of $\boldsymbol{R}$. $Disk(\text{merge}(\boldsymbol{R}, \sigma(\boldsymbol{S})))$ is the cost of one scan done from our algorithm and it is irrelevant

compared to the sorting. The dominant Disk I/O of our approach on the fact table $\boldsymbol{S}$ is therefore:

$$\begin{aligned} Disk(\boldsymbol{S}) \simeq & Disk(\text{read}(\boldsymbol{S})) + Disk(\text{sort}(\sigma(\boldsymbol{S})) \\ & = B_{\boldsymbol{S}} + \big(sel(\theta) * sel(\mathbf{G}) * B_{\boldsymbol{S}}\big) \log_M \big(sel(\theta) * sel(\mathbf{G}) * B_{\boldsymbol{S}}\big) \end{aligned}$$

1. In the best case, θ is always false, i.e., $sel(\theta) \to 0$, *or* all tuples in $\boldsymbol{R}$ belong to the same group, i.e., $sel(\mathbf{G}) \to 0$:

   $$sel(\theta) \to 0 \vee sel(\mathbf{G}) \to 0 \Leftrightarrow Disk(\text{sort}(\sigma(\boldsymbol{S})) = 0$$

   For $sel(\theta) \to 0$ *or* $sel(\mathbf{G}) \to 0$, the number of Disk I/Os on the fact table $\boldsymbol{S}$ is the linear lower bound i.e., $Disk(\text{sort}(\sigma(\boldsymbol{S}))) = 0 \Rightarrow Disk(\boldsymbol{S}) \simeq B_{\boldsymbol{S}}$.

2. In the worst case, all tuples satisfy predicate θ, i.e., $sel(\theta) = 1$, *and* there exists a tuple in $\boldsymbol{R}$ for any possible group of $\boldsymbol{S}$, i.e., $sel(\mathbf{G}) \to 1$.

   $$sel(\theta) \to 1 \wedge sel(\mathbf{G}) \to 1 \Leftrightarrow Disk(\text{sort}(\sigma(\boldsymbol{S})) = B_{\boldsymbol{S}} \log B_{\boldsymbol{S}}$$

   For $sel(\theta) \to 1$ *and* $sel(\mathbf{G}) \to 1$ the number of Disk I/Os on the fact table $\boldsymbol{S}$ is the upper bound, i.e., $Disk(\text{sort}(\sigma(\boldsymbol{S}))) = B_{\boldsymbol{S}} \log_M B_{\boldsymbol{S}} \Rightarrow Disk(\boldsymbol{S}) \simeq B_{\boldsymbol{S}} \log_M B_{\boldsymbol{S}}$.

□

Thus, opposite to current approaches which suffer from false hits and redundant fetches, the number of Disk I/Os on the fact table using our approach does not deteriorate in the presence of θ and $\mathbf{G}$. In fact, since $0 \leq sel(\theta) \leq 1$ and $0 \leq sel(\mathbf{G}) \leq 1$, our approach benefits from the predicate and group selectivities since the number of blocks to sort shrinks. Current solutions based on B-Tree indexes, instead, suffer since they have a $2|\boldsymbol{R}| \times \frac{1}{sel(\theta)}$ complexity, and, in the presence of a selective predicate, end up in fetching many blocks of the fact table for each single $\boldsymbol{R}$ tuple. Note that, in data warehouse scenarios, $sel(\theta)$ and $sel(\mathbf{G})$ are usually small since many different and heterogenous facts are stored in the fact table $\boldsymbol{S}$. For example, in the Swiss Feed Data Warehouse when a farmer asks for the protein content in the cereals, then, among all analyses stored in the data warehouse, only those storing a protein content (i.e., $N =$ 'CP') for the feeds he has grown (i.e., only the cereals) have to be considered. Finally, our approach does not require clustering: our approach fetches each block only once, independent of the number of

groups stored in it. Current solutions based on Segment Apply, instead, if a block stores tuples of m groups, fetch it m times.

Lemma 3. *(Memory) Let $\boldsymbol{R}$ be a relation with schema $\mathbf{R} \supseteq [\mathbf{G}, T]$, $\boldsymbol{S}$ be a fact table with schema $\mathbf{S} = [\mathbf{G}, T, \mathbf{A}]$, and θ be a predicate on $\mathbf{S}$. The number of memory I/Os on the fact table $\boldsymbol{S}$ of a Data Warehouse, for computing $\boldsymbol{R} \ltimes^{K2}_{\mathbf{G},\theta} \boldsymbol{S}$, is lower bounded by zero and is upper bounded by a linearithmic complexity.*

Proof. Similar to Lemma 1. We go directly to the memory I/O on the fact table, that is:

$$\begin{aligned} Mem.(\boldsymbol{S}) &\simeq Mem.(\text{read}(\boldsymbol{S})) + Mem.(\text{sort}(\sigma(\boldsymbol{S}))) \\ &= 0 + \left(sel(\theta) * sel(\mathbf{G}) * B_{\boldsymbol{S}}\right) \log_2 \left(sel(\theta) * sel(\mathbf{G}) * B_{\boldsymbol{S}}\right) \end{aligned}$$

Since the fact table $\boldsymbol{S}$ is initially read from disk, no memory I/O has to be counted for its first read. For the sorting, given B blocks, $B \log_M B$ I/Os are done on disk: $B \log_2 B - B \log_M B$ I/Os are done in main memory. Since $B \log_2 B - B \log_M B = B \log_2 B * (1 - \frac{1}{log_2 M}) \simeq B \log_2 B$, then the cost of sorting is given substituting B with the number of blocks to sort, i.e., $sel(\theta) * sel(\mathbf{G}) * B_{\boldsymbol{S}}$.

1. In the best case:

$$sel(\theta) \to 0 \vee sel(\mathbf{G}) \to 0 \Leftrightarrow Mem.(\text{sort}(\sigma(\boldsymbol{S}))) = 0$$

   For $sel(\theta) \to 0$ *or* $sel(\mathbf{G}) \to 0$, the number of Memory I/Os on the fact table $\boldsymbol{S}$ is the lower bound, which is zero, since no sorting is computed, i.e., $Mem.(\text{sort}(\sigma(\boldsymbol{S}))) = 0 \Rightarrow Mem.(\boldsymbol{S}) = 0$.

2. In the worst case:

$$sel(\theta) \to 1 \wedge sel(\mathbf{G}) \to 1 \Leftrightarrow \textit{Mem}(\text{sort}(\sigma(\boldsymbol{S}))) = B_{\boldsymbol{S}} \log_2 B_{\boldsymbol{S}}$$

   For $sel(\theta) \to 1$ *and* $sel(\mathbf{G}) \to 1$ the number of Memory I/Os on the fact table $\boldsymbol{S}$ has its upper bound, i.e., $Mem.(\text{sort}(\sigma(\boldsymbol{S})) = B_{\boldsymbol{S}} \log B_{\boldsymbol{S}} \Rightarrow Mem.(\boldsymbol{S}) = B_{\boldsymbol{S}} \log B_{\boldsymbol{S}}$.

□

The memory I/O of our approach benefits from the predicate selectivity and the group selectivity, since $0 \leq sel(\theta) \leq 1$ and $0 \leq sel(\mathbf{G}) \leq 1$. It is independent of the overlapping of the groups in the blocks. In the best case, our approach has no memory I/O at all.

Lemma 4. *(CPU) Let $\boldsymbol{R}$ be a relation with schema $\mathbf{R} \supseteq [\mathbf{G}, T]$, $\boldsymbol{S}$ be a fact table with schema $\mathbf{S} = [\mathbf{G}, T, \mathbf{A}]$, and θ be a predicate on $\mathbf{S}$. The number of CPU operations on the fact table $\boldsymbol{S}$ of a Data Warehouse, for computing $\boldsymbol{R} \ltimes^{T} [\mathbf{G}, \theta]\, \boldsymbol{S}$, is lower bounded by a linear complexity and is upper bounded by a linearithmic complexity.*

Proof. (CPU) From the tree of Figure 2.4(a), the cpu costs of each operation are:

$$
\begin{aligned}
&\mathit{CPU}(\sigma(\boldsymbol{S})) \simeq |\boldsymbol{S}| \\
&\mathit{CPU}(\text{sort}(\boldsymbol{R})) \simeq |\boldsymbol{R}| \log_2 |\boldsymbol{R}| \\
&\mathit{CPU}(\text{sort}(\sigma(\boldsymbol{S})) \simeq \big(sel(\theta) * sel(\mathbf{G}) * |\boldsymbol{S}|\big) \log_2 \big(sel(\theta) * sel(\mathbf{G}) * |\boldsymbol{S}|\big) \\
&\mathit{CPU}(\text{merge}(\boldsymbol{R}, \sigma(\boldsymbol{S}))) \simeq |\boldsymbol{R}| + sel(\theta) * sel(\mathbf{G}) * |\boldsymbol{S}|
\end{aligned}
$$

For the selection $\sigma(\boldsymbol{S})$, each tuple of $\boldsymbol{S}$ must be evaluated against condition $\mathbf{G} \in \pi_{\mathbf{G}}(\boldsymbol{R}) \wedge \theta$. The SortMerge procedure consists of a sort and of a merge procedure. The merging is negligible compared to the sorting. All CPU costs are approximated (symbol $\simeq$) since each CPU operation for sorting or merging has actually cost 3: 1 for comparing the grouping attribute, 1 for comparing the similarity attribute, and 1 for the logical $\wedge$ between the two comparisons. The dominant CPU cost on the fact table $\boldsymbol{S}$ is:

$$
\begin{aligned}
\mathit{CPU}(\boldsymbol{S}) \simeq& \mathit{CPU}(\sigma(\boldsymbol{S})) + \mathit{CPU}(\text{sort}(\boldsymbol{S})) \\
\simeq& |\boldsymbol{S}| + \big(sel(\theta) * sel(\mathbf{G}) * |\boldsymbol{S}|\big) \log_2 \big(sel(\theta) * sel(\mathbf{G}) * |\boldsymbol{S}|\big)
\end{aligned}
$$

1. In the best case:

$$sel(\theta) \to 0 \vee sel(\mathbf{G}) \to 0 \Leftrightarrow \mathit{CPU}(\text{sort}(\sigma(\boldsymbol{S}))) = 0$$

   For $sel(\theta) \to 0$ *or* $sel(\mathbf{G}) \to 0$, the number of CPU operations on the fact table $\boldsymbol{S}$ has its lower bound i.e., $\mathit{CPU}(\text{sort}(\sigma(\boldsymbol{S}))) = 0 \Rightarrow \mathit{CPU}(\boldsymbol{S}) \simeq |\boldsymbol{S}|$, which is linear.

2. In the worst case:

$$sel(\theta) \to 1 \wedge sel(\mathbf{G}) \to 1 \Leftrightarrow \mathit{CPU}(\text{sort}(\sigma(\boldsymbol{S}))) = |\boldsymbol{S}| \log |\boldsymbol{S}|$$

For $sel(\theta) \to 1$ *and* $sel(\mathbf{G}) \to 1$ the number of CPU operations on the fact table $\boldsymbol{S}$ has its upper bound, i.e., $CPU(\text{sort}(\sigma(\boldsymbol{S}))) = |\boldsymbol{S}| \log_2 |\boldsymbol{S}| \Rightarrow CPU(\boldsymbol{S}) \simeq |\boldsymbol{S}| \log_2 |\boldsymbol{S}|$.

□

Example 6. Using the example of Figure 2.2, we use the Swiss Feed Data Warehouse to compute $\boldsymbol{R} \ltimes^T [G, N =$ 'CP' $\wedge R > 0.7]\ \boldsymbol{S}$ with the following parameters: 20 groups in $\boldsymbol{R}$ (i.e., the number of cereals); a fact table $\boldsymbol{S}$ with $B_{\boldsymbol{S}} = 10$M blocks and $|\boldsymbol{S}| = 1G$ tuples; the predicate selectivity $sel(N =$ 'CP' $\wedge\ R > 0.7)$ is 3% (i.e., among all lab analysis, 3% corresponds to crude protein measurements and has a reliability greater than 0.7); the group selectivity $sel(\mathbf{G})$ is 5% (i.e., among all possible animal feeds, 5% are cereals); $mem_size = 4$k (i.e., the default buffer size in PostgreSQL in terms of number of blocks). With these numbers we get:

$$\text{Disk}(\boldsymbol{S}) \simeq 15k \log_{4k} 15k + 10M \simeq 10M$$
$$\text{Mem}(\boldsymbol{S}) \simeq 15k \log_2 15k \simeq 200k$$
$$\text{CPU}(\boldsymbol{S}) \simeq 1.5M \log_2 1.5M + 1G \simeq 1G$$

As the reader can see from Example 2, the main cost for our approach is the access to the fact table. The sorting has only a low impact, because our approach takes advantage from the predicate and the group selectivities. This can be easily verified in our experiments in Figure 2.8.c and Figure 2.8.d where, respectively, for any predicate selectivity and any group selectivity below 20%, the $roNNJ$ stays stable.

2.7 Experiments

This section empirically compares our approach with the state of the art techniques for computing NNJs. We consider: *i)* the AntiJoin [ME92] (AntiJ) that, for a pair (r,s) checks via a NOT EXISTS subquery that no closer tuple t with the same group exists in $\boldsymbol{S}$; *ii)* the B-Tree [YLK10] (B-Tree), using indexed MIN and MAX subqueries; *iii)* the SegmentApply [SAA10],[GLJ01] (SegApply), implemented as a Lateral query running multiple (one per group) group-unaware SortMerge NNJs; *iv)* the robust NNJ (roNNJ). We compute NNJs in the Swiss Feed Data Warehouse and on a TPC-H dataset. As for our running example, we use the animal feed as grouping attribute and the time as similarity attribute. In the average case, we have 20 groups in $\boldsymbol{R}$ (i.e., the number of different cereals in the Swiss Feed Data Warehouse), $|\boldsymbol{R}| = 60$k (since we have

3k timestamps per feed on average), $|\boldsymbol{S}| = 1$G, a predicate with selectivity $sel(\theta) = 0.05$, and an index on the grouping attribute. We vary each of those variables and show how the approaches behave. Our experiments show that: *i*) the B-Tree is the most sensitive technique to the cardinality of $\boldsymbol{R}$ (since the number of look-ups to perform is $2|\boldsymbol{R}|$), and to $sel(\theta)$ (since for each look-up $\frac{1}{sel(\theta)}$ false hits are computed); *ii)* SegApply is the most sensitive technique to the size of $\boldsymbol{S}$ (since the number of blocks that are redundantly fetched grows) and to $sel(\mathbf{G})$ (since the number of times the blocks are redundantly fetched grows); *iii)* while current solutions are competitive on column-store DBMSs only if the fact table is clustered on (G,T), our approach is efficient even without clustering; *iv)* $roNNJ$is the only technique that does not deteriorate if an index on the groups is not available.

Due to its high runtime, the AntiJoin will just be shown in scenarios where its performances are competitive. All approaches are implemented using PostgreSQL 9.3.4, Apache Cassandra 3.4, and MonetDB 11.21.5.

For the experiments on disk, we used a 2.66 GHz Intel Core i7 machine with 4GB main memory and a 480 GB Solid State Drive, running Mac OS X 10.9. The PostgreSQL cache (*shared_buffers* parameter) and the memory used for sorting (*work_mem* parameter) have been set to their default value, i.e., respectively, 32MB and 1MB.

For the experiments in main memory, we used a 2 x Intel(R) Xeon(R) CPU E5-2440 (6 cores each) @ 2.40GHz with 64GB main memory, and running CentOS 6.4 (L1 cache: 192 KB, L2 cache: 1536 KB, L3 cache: 15360 KB). The PostgreSQL cache has been set to 10 GB and the memory used for sorting is 10 GB. All indices and all data are kept in memory and no disk I/O for reading or sorting is done.

For the experiments on column-store DBMSs, we use MonetDB and Cassandra. Specifically, MonetDB has been used for the computation of SegApply (Cassandra does not support subqueries), while Cassandra for the computation of AntiJ and B-Tree (MonetDB is not robust in computing AntiJ and starved our machines of memory; it never takes advantage of the B-Tree for computing Min/Max queries).

2.7.1 Scalability on Disk

Figure 2.8 shows that $roNNJ$ is the most robust technique on disk since it fetches each block of the fact table only once, independently on the predicate and group selectivities.

The AntiJ (Figure 2.8.a) quickly deteriorates since it first builds all (r, s) pairs with the same group, and then checks in $\boldsymbol{S}$ that no closer tuple than s exists. This has a cubic complexity, and is efficient only when the predicate (Figure 2.8.c) selects very few tuples (e.g., $sel(\theta) = 10^{-6}\%$ selects only one tuple in our scenario).

Although the B-Tree does not require any sorting and its performances are almost independent on the size of the fact table (Figure 2.8.b), it becomes extremely inefficient in the presence of a selective predicate (Figure 2.8.c). Furthermore, it does not scale well if the size of $\boldsymbol{R}$ grows (Figure 2.8.a), and becomes 1 order of magnitude slower than $roNNJ$for more than 200k outer tuples. For this approach, the number of index look-ups to compute does not depend on the number of groups stored in $\boldsymbol{R}$, but just on its cardinality. However (Figure 2.8.d), with few groups the index false hits point always to the same blocks that, once fetched, are cached in memory; with many relevant groups, instead, the false hits fetch different blocks and the runtime increases.

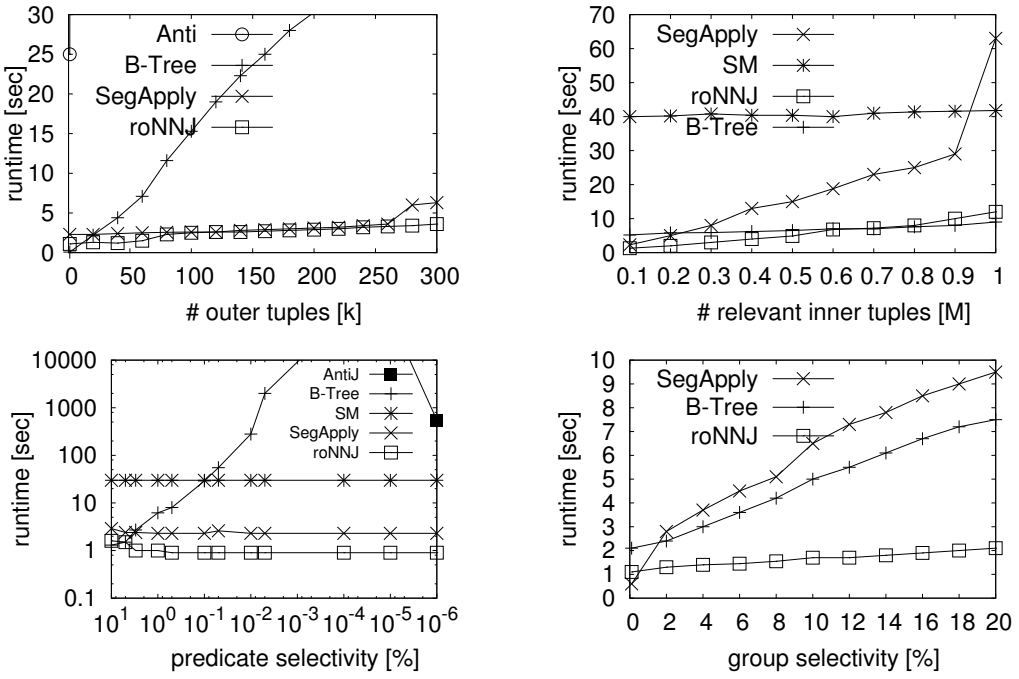


Figure 2.8: Scalability on Disk by varing the size of $\boldsymbol{R}$ (a), the size of $\boldsymbol{S}^{G,\theta}$ (b), the predicate selectivity (c), and the group selectivity (d).

Figure 2.8.a shows that, between 0 and 260k outer tuples, SegApply is stable since the main cost (fetching and sorting the relevant blocks of the fact table) stays the same. The jump after 260k outer tuples is because the sorting of $\boldsymbol{R}$ is done first in main memory using Quicksort,

then on disk using external sorting. The same happens after 60k tuples for $roNNJ$, because it sorts the $\boldsymbol{R}$ tuples all together (and not just one segment at the time). The reader can see in Figure 2.8.b that SegApply does not scale well when the size of the fact table increases, since the number of blocks to fetch (redundantly) increases. For more than 900k inner tuples, the blocks to (redundantly) read cannot all be cached in memory anymore, and have to be refetched from disk. In Figure 2.8.d we show that, the higher the group selectivity, the higher is the number of times that the blocks of the fact table are redundantly read.

For completeness, in Figure 2.8.b and Figure 2.8.c, we show that non-indexed approaches (such as SortMerge, *SM*) are slower than their indexed counterpart since they fetch *all* blocks of $\boldsymbol{S}$ rather than just the ones storing tuples with the groups of $\boldsymbol{R}$.

2.7.2 Scalability on Main Memory

Also in main memory, $roNNJ$is the most robust approach. For an in-memory execution, Quicksort is used by both $roNNJ$and SegApply, and no jump in the runtime occur when the size of $\boldsymbol{R}$ increases (Figure 2.9.a). When the number of inner tuples increases (Figure 2.9.b) SegApply

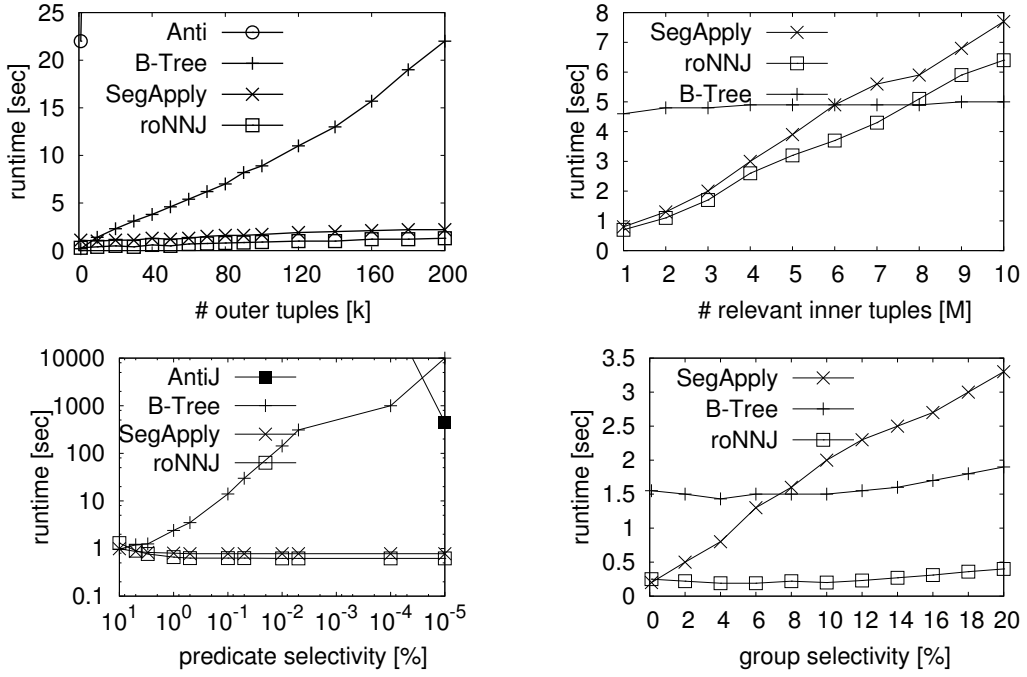


Figure 2.9: Scalability on Memory by varying the size of $\boldsymbol{R}$ (a), the size of $\boldsymbol{S}^{G,\theta}$ (b), the predicate selectivity (c), and the group selectivity (d).

reduces its gap to $roNNJ$ since the latency of reading redundantly the blocks of $\boldsymbol{S}$ from memory is smaller than the one from disk. However it still suffers from redundant memory fetches when the group selectivity increases (Figure 2.9.d).

In Figure 2.9.c we show that, in the presence of an extremely selective predicate (e.g., $sel(\theta) = 0.000001$ selects only one tuple in our scenario) the AntiJoin performs better than the B-Tree because it fetches from memory each relevant block $|\boldsymbol{R}| = 60$k times rather than $\frac{1}{0.000001} = 1$M times. However, the reader can see in Figure 2.9.d that, opposite to the experiments on disk, the runtime of the B-Tree is pretty stable when the number of groups to process increases: it is constant up to a group selectivity of 12%, and it then slightly increases since the buffering effect of the operating systems has less impact when many groups are processed.

2.7.3 Scalability Without Indexes Availability

In this subsection we evaluate the approaches when no index is available on the grouping attribute, and we show the state of the art solutions are two order of magnitude slower than $roNNJ$since they are not group-enabled.

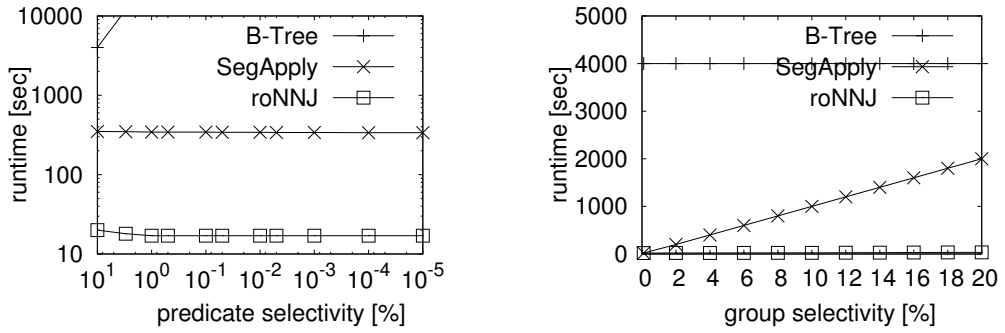


Figure 2.10: Scalability when an index on $\mathbf{G}$ is not available, varying the predicate selectivity (left) and group selectivity (right).

In Figure 2.10(a), SegApply must performs 20 scans on the fact table (since 20 groups are stored in $\boldsymbol{R}$), and is thus 20 times slower than $roNNJ$. When the group selectivity increases, the number of scans of the fact table increases, too, making the approach inefficient.

When only an index on T is available, the B-Tree checks the equality on $\mathbf{G}$ similar to the selection θ. This increases the number of index false hits computed. If the closest tuple fetched does not

have the same group as the outer tuple, a false hit has been computed, and the index must be scanned until a tuple with the same group satisfying θ is found. This is extremely inefficient.

2.7.4 Scalability in Column-Store DBMSs

In Figure 2.11.a, we show that the B-Tree is competitive only for a small $\boldsymbol{R}$ relation (e.g., less than 1k tuples). This is so because, when a secondary index is present, Cassandra finds the nearest neighbour of r by first selecting in the fact table *all* the tuples of group $r.\mathbf{G}$, and then computing the Min and Max on the selected tuples. Retrieving all the tuples of the same group (in order to compute the distances) makes this approach slow for a large outer relation. However, in the presence of a very selective predicate (Figure 2.11.c), such an approach takes advantage of the predicate selectivity and performs opposite to its row-store counterpart. It avoids the false hits since it scans each column involved in θ and on the fly filters out all the entries not satisfying it. The approach is, at its best, 2 order of magnitude slower then $roNNJ$.

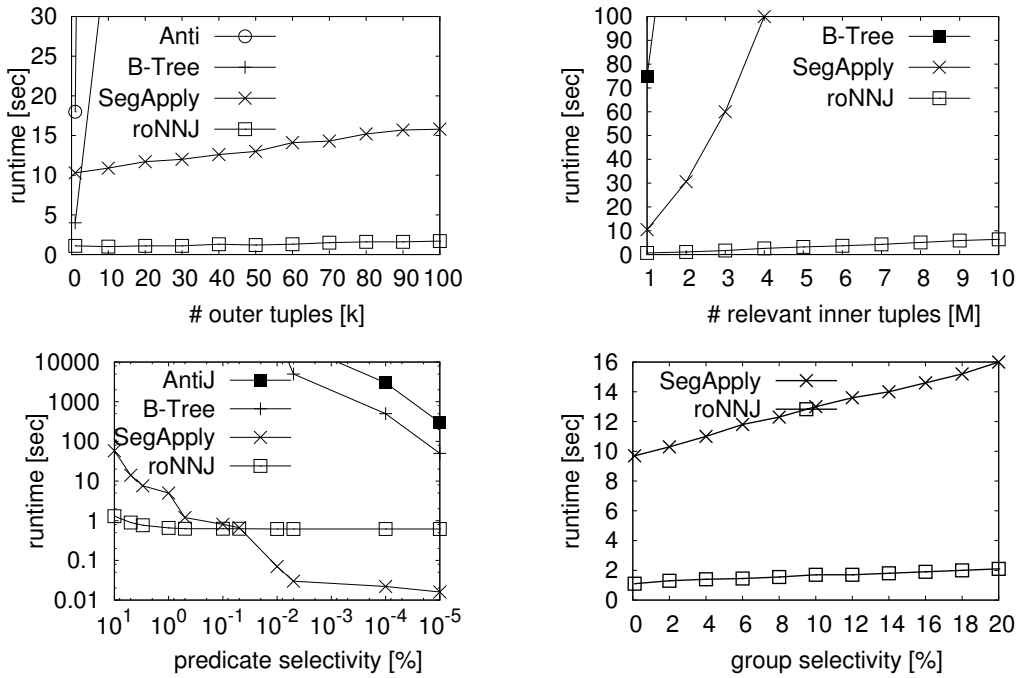


Figure 2.11: Scalability in Column-Stores by varying the size of $\boldsymbol{R}$ (a), the size of $\boldsymbol{S}^{G,\theta}$ (b), the predicate selectivity (c), and the group selectivity (d).

Similarly, AntiJ quickly deteriorates since it has a cubic complexity even if just $\mathbf{G}$ and T have to be accessed for computing the distances.

By comparing Figure 2.11.a with Figure 2.9.a, the reader can see that SegApply suffers much more when it's implemented in column-store DBMSs because every column accessed before the NNJ (i.e., $\mathbf{G}$, T, and the ones involved in θ) is affected by redundant fetches. In our experiments redundant fetches are repeated 3 times: for the grouping attribute (*feed_name*) and for the attributes involved in θ (*nutrient_name* and *reliability*). Furthermore, by comparing Figure 2.11.b with Figure 2.9.b, the reader can see that, since a block stores much more (OID, Value) pairs than tuples, the probability that a block stores data of different groups is much higher in column-store than in row-store databases, and much more redundant fetches are computed. However, in the presence of a very selective predicate (Figure 2.11.c), SegApply applies early materialization and, by fetching T only for the (few) entries satisfying θ, speeds up the NNJ.

2.7.5 Scalability on a Clustered Fact Table

In this subsection we evaluate the approaches in the atypical scenario when the a a primary index is available, i.e., when the fact table is clustered by $(\mathbf{G}, T)$. This is rarely the case in real world applications, since the grouping and similarity attributes change for different queries. In Figure 2.12.a we show that when a primary index is available, the B-Tree suffers less compared

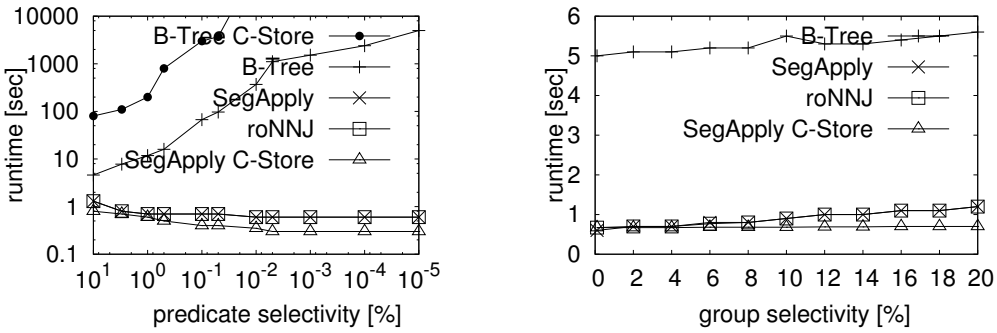


Figure 2.12: Scalability on a Clustered Fact Table, by varying the predicate selectivity (left) and group selectivity (right).

to Figure 2.8.a because the data is clustered: and each index false hit happen on the same or on the next block that, once fetched, is cached in main memory. The approach[7] stays however not

[7]For the experiment on column-store DBMSs, we use a clustered B-Tree in Apache Cassandra, and we rewrite the Min(T)/Max(T) queries as ORDER BY T LIMIT 1 statements. MonetDB does not optimize MIN(T)/MAX(T) queries when a predicate θ is present, even if a primary index on T is available. It is therefore less efficient than Cassandra in computing NNJs with a B-Tree since, for a given $r \in \boldsymbol{R}$, it always fetches all tuples from $\boldsymbol{S}$ with group $r.\mathbf{G}$.

competitive compared to $roNNJ$. SegApply performs the same as $roNNJ$since no redundant fetches are computed (most of the blocks store tuples of exactly one group). SegApply implemented on column-store DBMSs is even faster than $roNNJ$, for two reasons: *i)* when the fact table is clustered by $(\mathbf{G}, T)$ no redundant fetches are done; *ii)* the value of *every* attribute (except T) is fetched after the NNJ in sort-order, i.e., with just a scan of the columns.

2.7.6 Real World Queries in the Swiss Feed Data Warehouse

In this subsection we evaluate how the approaches compute three queries $\mathbf{q_0}$, $\mathbf{q_1}$, and $\mathbf{q_2}$ computing *derived nutrients* in the Swiss Feed Data Warehouse. Derived nutrients are computed applying a formula expression on the result of a sequence of NNJs (cf. Section 5.3): $\mathbf{q_0}$, with selectivity $sel(\theta) = 0.1$ and with two NNJs, calculates the *Gross Energy* value; $\mathbf{q_1}$, with $sel(\theta) = 0.05$ and with two NNJs calculates the *Degradability of Proteins*; $\mathbf{q_2}$, with $sel(\theta) = 0.1$ and with five NNJs, calculates the *Absorbable Proteins*. For each of those queries, 20k $\boldsymbol{R}$ tuples of 3 different feeds (groups) have been used; the predicate θ is not important since, independently from the condition itself, only its selectivity $sel(\theta)$ influences the runtime (in row-store DBMSs) of the approaches.

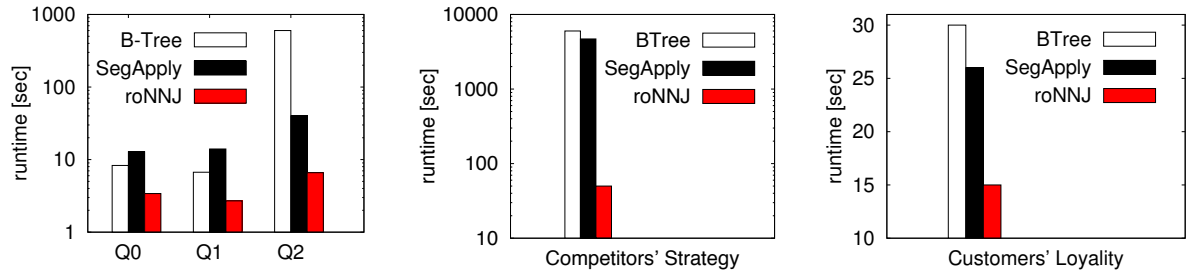


Figure 2.13: *Left* Chart: Top Queries in the Swiss Feed Data Warehouse: $\mathbf{q_0}$ ($sel(\theta) = 0.1$, 2 NNJs), $\mathbf{q_1}$ ($sel(\theta)$=0.05, 2 NNJs), $\mathbf{q_2}$($sel(\theta)$=0.1, 5 NNJs). *Middle* and *Right* charts: Top Queries in TPC-H: Competitors' Strategy ($sel(\theta)$ = 0.04,1 NNJ), Customers' Loyality ($sel(\theta)$ = 0.04, 1 NNJ)

The left plot of Figure 2.13 shows that $\mathbf{q_2}$ is the query taking longest among the three. This is due to the higher number of joins (5 NNJs) computed. The B-Tree is two order of magnitude slower than $roNNJ$ since, after each NNJ, the number of outer tuples for the next NNJ of the sequence gets bigger due to multiple join matches: the number of index look-ups to compute compared to the previous join also grows. Comparing $\mathbf{q_0}$ with $\mathbf{q_1}$ (same number of NNJs but $sel(\theta)$ reduced from 0.1 to 0.05), we see that the B-Tree and $roNNJ$ become slightly better: the former since

the number of result tuples of the first join decreases (this determines the number of outer tuples for the second join of the sequence), the latter since the number of relevant tuples to sort shrinks. Overall $roNNJ$ is the fastest for all three queries because it does not fetch blocks redundantly and it does not suffer from θ.

2.7.7 Real World Queries in TPC-H

The middle and right plots of Figure2.13 evaluate the approaches using a TPC-H dataset. A Fact Table with 83M LineItems has been generated. This corresponds to orders of 400k different customers of 700k products of 25 different brands from 25 different suppliers. The following queries had been asked:

Competitors' Strategy (similarity on the product quantity): For each order to German suppliers, compare the discount that Chinese suppliers make for the same product (*p_partkey*) on the order with the most similar product quantity (*l_quantity*):

- $\mathbf{G}$ = *p_partkey*, T = *l_quantity*
- $|\boldsymbol{R}| = 3$M, i.e., the orders made to german suppliers
- $sel(\theta) = 1/25$, i.e., $\theta \equiv n\_name =$ 'China' selects 4% of the tuples (tuples are uniformly distributed with regards to their supplier's nation)
- $|\pi_{p\_partkey}(\boldsymbol{R})| = 100k$, i.e., 100k relevant groups

Customers' Loyality (similarity on the time): For each order of the top 400 customers (*l_custkey*) of Brand#41, give the items of Brand#21 they have bought in the closest order (*l_commitdate*):

- $\mathbf{G}$ = *l_custkey*, T = *l_commitdate*
- $|\boldsymbol{R}| = 4k$, i.e., on average a top customer has 10 orders with products of Brand#41
- $sel(\theta) = 1/25$, i.e., $\theta \equiv p\_brand =$ 'Brand#21' selects 4% of the tuples (tuples are uniformly distributed in regards to the brands)
- $|\pi_{l\_custkey}(\boldsymbol{R})| = 400$, i.e., 400 relevant groups

The B-Tree performs worst because of a highly selective θ and a high number of $\boldsymbol{R}$ tuples: for the first top query (middle plot of Figure 2.13.a), 3M $\times$ 25 index false hits are computed. Also SegmentApply performs the first query very slowly because of the redundant fetches due to the high (i.e., $100k$) number of relevant groups. $roNNJ$ performs best for both queries because it access the fact table only once and does not compute index false hits.

2.8 Conclusion and Future Work

In this work we have introduced a new algebraic operator: the group- and selection-enabled Nearest Neighbour Join. Its evaluation query tree is not dependent on the physical organization of the fact table, and, opposite to the state of the art solutions, does not suffer from index false hits and redundant fetches. We have described the implementation of our query tree both in row-store and in column-store DBMSs. We have shown that, opposite to the state of the art solutions, a group- and selection-enabled query tree enlarges the scope of the query optimizer, which can take full advantage of the optimizations on the groups and on the predicate. We have implemented an efficient algorithm, $roNNJ$, that computes the NNJ in a single scan of the input relations. We have analytically shown that our approach is upper bounded by a complexity of $n \log n$. As future work, we intend to introduce a NNJ that computes the similarity for timestamps with different granularities: in the Swiss Feed Data Warehouse, for some measurements, only the month, the season or the year are available instead of the full date. We also intend to apply our findings in nearest neighbour joins with queries with user-defined distance functions.

Acknowledgments

This work has been developed in collaboration with Agroscope Switzerland in the context of the Tameus project, with funding from the Swiss National Science Foundation.

CHAPTER 3

Disjoint Interval Partitioning

Abstract

In databases with time interval attributes, query processing techniques that are based on sort-merge or sort-aggregate, deteriorate. This happens because for intervals no total order exists and either the start or end point is used for the sorting. Doing so leads to inefficient solutions with lots of *unproductive comparisons* that do not produce an output tuple. Even if just one tuple with a long interval is present in the data, the number of unproductive comparisons of sort-merge and sort-aggregate gets quadratic.

In this Chapter we propose $\mathcal{DIP}$ (Disjoint Interval Partitioning), a technique to efficiently perform sort-based operators on interval data. $\mathcal{DIP}$ divides an input relation into the minimum number of partitions, such that all tuples in a partition are non-overlapping. The absence of overlapping tuples guarantees efficient sort-merge computations without backtracking. With $\mathcal{DIP}$ the number of unproductive comparisons is linear in the number of partitions. In contrast to current solutions with inefficient random accesses to the active tuples, $\mathcal{DIP}$ fetches the tuples

in a partition sequentially. We illustrate the generality and efficiency of $\mathcal{DIP}$ by describing and evaluating three basic database operators over interval data: join, anti-join, and aggregation.

3.1 Introduction

Many databases model real-world states that change. To model state changes the most common approach is to associate each tuple with a time interval $T = [T_s, T_e)$ that represents the time period during which the tuple is valid [DDL03]. In this Chapter we propose an efficient technique to perform sort-based computations over temporal relations, i.e., relations with an interval attribute. For example, the temporal relations in Figure 3.1 record the bookings of luxury suites at hotels $\boldsymbol{R}$ and $\boldsymbol{S}$, where T is the booking period of room # at price \$.

$\boldsymbol{R}$	T	#	\$
r_1	$[1,5)$	1	80
r_2	$[6,8)$	1	60
r_3	$[7,8)$	2	80
r_4	$[7,10)$	3	75
r_5	$[10,11)$	2	70
r_6	$[10,11)$	5	80

$\boldsymbol{S}$	T	#	\$
s_1	$[0,8)$	6	60
s_2	$[1,2)$	2	70
s_3	$[3,4)$	2	80
s_4	$[5,6)$	3	60
s_5	$[9,12)$	2	90
s_6	$[11,12)$	1	90

Figure 3.1: Temporal relations $\boldsymbol{R}$ and $\boldsymbol{S}$

Techniques based on sorting have a long tradition in DBMSs and are used extensively by the query evaluation engine. Specifically, sort-merge is used for joins, anti-joins and nearest neighbour joins [CBB15b], whereas sort-aggregate is used for aggregations and duplicate elimination [Gra93]. Consider a temporal join where tuples $r_i \in \boldsymbol{R}$ and $s_j \in \boldsymbol{S}$ shall be joined iff their intervals overlap. To ensure that all join matches for an outer tuple r_{i+1} are found, sort-merge must backtrack in the inner relation to the first tuple $s_k \in \boldsymbol{S}$ that overlaps with tuple r_i. This is equivalent to the handling of non-key attributes in sort-merge joins [LGS02], but the crucial difference when dealing with T is that the join matches for a tuple $r \in \boldsymbol{R}$ in relation $\boldsymbol{S}$ are *non-consecutive*, and many non-matching tuples might have to be rescanned. This makes sort-merge inefficient for interval data.

Example 7. To compute a temporal join using sort-merge, $\boldsymbol{R}$ and $\boldsymbol{S}$ are sorted by start point T_s, and then processed as illustrated in Figure 3.2. The middle part illustrates the pairs of tuples that must be compared. First, tuple r_1 is compared with s_1. The tuples are joined since $[1,5)$ overlaps

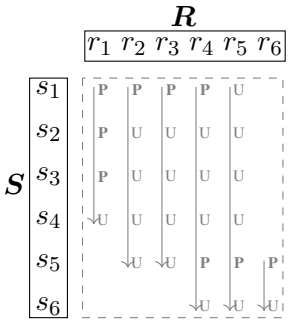


Figure 3.2: Temporal join using sort-merge: for $r_{i+1} \in \boldsymbol{R}$, backtracking must go back in $\boldsymbol{S}$ to the first join match of r_i.

with $[0, 8)$. We proceed with the tuples from $\boldsymbol{S}$ until we fetch a tuple that starts after r_1 ends. Thus, no tuples after s_4 must be looked at. Next, tuple r_2 is fetched and we must *backtrack* in $\boldsymbol{S}$. To ensure that all join matches for r_2 are found, we must go back to the first join match of r_1 (i.e., s_1), and compare tuple r_2 with s_1, s_2, s_3, s_4, and s_5. Similarly all other tuples in $\boldsymbol{R}$ are processed. Observe that r_4 joins with non-consecutive tuples in $\boldsymbol{S}$: it joins with s_1, does not join with s_2, s_3 and s_4, and again joins with s_5.

A comparison that does not produce a match is an *unproductive* comparison. Example 7 shows that a sort-merge join may perform many more unproductive (U) than productive (P) comparisons. To limit the amount of unproductive comparisons in sort-merge computations, we propose $\mathcal{DIP}$ (*Disjoint Interval Partitioning*). $\mathcal{DIP}$ partitions an input relation into the smallest possible number of partitions, each storing tuples with non-overlapping time intervals. Figure 3.3

$\boldsymbol{R}_1$	T	#	$
r_1	$[1, 5)$	1	80
r_2	$[6, 8)$	1	60
r_6	$[10, 11)$	5	80

$\boldsymbol{S}_1$	T	#	$
s_1	$[0, 8)$	6	60
s_6	$[11, 12)$	1	90

$\boldsymbol{R}_2$	T	#	$
r_3	$[7, 8)$	2	80

$\boldsymbol{S}_2$	T	#	$
s_2	$[1, 2)$	2	70
s_3	$[3, 4)$	2	80
s_4	$[5, 6)$	3	60
s_5	$[9, 12)$	2	90

$\boldsymbol{R}_3$	T	#	$
r_4	$[7, 10)$	3	75
r_5	$[10, 11)$	2	70

Figure 3.3: *CreateDIP*($\boldsymbol{R}$) and *CreateDIP*($\boldsymbol{S}$)

shows the result of $\mathcal{DIP}$ applied to our example relations. The partitioning yields three outer

and two inner $\mathcal{DIP}$ partitions. Note that tuples of different partitions may overlap, but inside a single partition tuples do not overlap. Thus, a *$\mathcal{DIP}$Merge* between the partitions does not have to backtrack. Moreover, since $\mathcal{DIP}$ produces partitions with tuples that are sorted, no additional sorting is required prior to computing a *$\mathcal{DIP}$Merge*.

Example 8. Figure 3.4 illustrates the computation of the temporal join. Two *$\mathcal{DIP}$Merge*

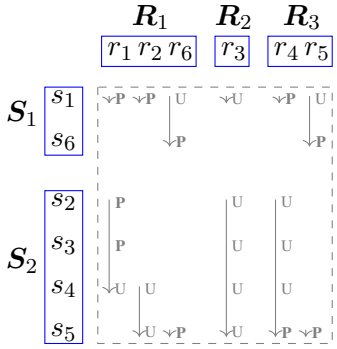


Figure 3.4: *$\mathcal{DIP}$Merge* between two $\mathcal{DIP}$ partitions is performed without backtracking.

steps are computed: *$\mathcal{DIP}$Merge*$(\{\boldsymbol{R}_1, \boldsymbol{R}_2, \boldsymbol{R}_3\}, \boldsymbol{S}_1, \bowtie)$ joins $\{\boldsymbol{R}_1, \boldsymbol{R}_2, \boldsymbol{R}_3\}$ with $\boldsymbol{S}_1$, and *$\mathcal{DIP}$Merge*$(\{\boldsymbol{R}_1, \boldsymbol{R}_2, \boldsymbol{R}_3\}, \boldsymbol{S}_2, \bowtie)$ joins $\{\boldsymbol{R}_1, \boldsymbol{R}_2, \boldsymbol{R}_3\}$ with $\boldsymbol{S}_2$. For a single *$\mathcal{DIP}$Merge* each input partition is scanned just once. For example, for the first *$\mathcal{DIP}$Merge*, tuple r_1 is compared with s_1, producing a join match. Since r_1 ends before s_1, we advance $\boldsymbol{R}_1$ and fetch r_2 producing a second join match. Tuple r_6 is fetched next and unproductively compared to s_1. Since r_6 ends after s_1 we are sure that in $\boldsymbol{R}_1$ we have found all tuples overlapping s_1. We therefore switch to partition $\boldsymbol{R}_2$ (and later to $\boldsymbol{R}_3$), which is processed similarly. After the tuples overlapping s_1 have been found in all outer partitions, we fetch s_6 from $\boldsymbol{S}_1$ and resume the scan of $\boldsymbol{R}_1$ from where it stopped (i.e., r_6): no backtracking is necessary.

$\mathcal{DIP}$ guarantees that the number of unproductive comparisons is linear in the number of partitions, i.e., is upper-bounded by $c \times n$, where c is the number of partitions and n is the number of tuples. The number of partitions is the maximum number of tuples in a relation that overlap *contemporaneously*. While sort-merge becomes quadratic as soon as *one* long lived tuple exists in a relation, for $\mathcal{DIP}$ *all* tuples in a relation must overlap to make it quadratic.

Existing partitioning techniques segment the time domain and place the tuples into segments they overlap [CGN+14]. Various research questions have been tackled in this context. Among others, disjoint segments [SSJ94], overlapping segments [DBG14], variable-size segments [DBG14], and the replications of tuples in all segments they overlap [LM92] have been investigated. In all

cases the (implicit) goal has been to place tuples with similar intervals into the same partitions. $\mathcal{DIP}$ does exactly the opposite: it puts tuples that do not overlap into the same partition. This yields more merges between partitions, but the merges no longer require a nested-loop and are performed much more efficiently: in n rather than n^2 time.

Our approach is general, simple and systematic: to compute a temporal join, anti-join, or aggregation, we first compute $\mathcal{DIP}$ on the input relations, and then apply a sequence of $\mathcal{DIP}$*Merge*s on the partitions. In our experiments we show that $\mathcal{DIP}$, despite its generality, manages data histories much more efficiently than the more specialized state-of-the-art solutions. The number of partitions is independent of the length of the history, and there is only a linear dependency between the runtime and the size of partitions. We show that current solutions, such as the Timeline Index [KMV+13] or the Sweepline [APR+98] algorithm, incur less unproductive comparisons but are slower than $\mathcal{DIP}$ since they suffer from random (disk or memory) accesses: the Timeline Index since it computes one index look up for each matching tuple; Sweepline since after a series of insertions and deletions in the list of active tuples, the elements of the list become randomly scattered in memory [PHD16].

Our technical contributions are as follows:

1. We propose the *Create*$\mathcal{DIP}(\boldsymbol{R})$ algorithm to partition a relation $\boldsymbol{R}$ into the smallest number of $\mathcal{DIP}$ partitions with non-overlapping tuples. We prove that the number of partitions is minimal.

2. We show how to use $\mathcal{DIP}$ partitions to compute temporal joins, anti-joins, and aggregations. We prove that the number of unproductive comparisons per tuple is upper-bounded by the number of $\mathcal{DIP}$ partitions.

3. We introduce an efficient algorithm, $\mathcal{DIP}$*Merge*, to efficiently compute a temporal join, anti-join, and aggregation over multiple $\mathcal{DIP}$ partitions with one sequential scan of the input partitions and no backtracking.

4. We experimentally show that $\mathcal{DIP}$ is the only technique that, either with disk- or memory-resident data, computes temporal joins, anti-joins, and aggregations without deteriorating if the data history grows.

The rest of the Chapter is organized as follows. Section 2 discusses related work. After the background in Section 3, we present Disjoint Interval Partitioning ($\mathcal{DIP}$) in Section 4 and its

implementation in Section 5. Section 6 quantifies the costs for, respectively, a temporal join, anti-join, and aggregation using $\mathcal{DIP}$. Section 7 describes the implementation of $\mathcal{DIP}Merge$. Section 8 reports the results of our empirical evaluation. Section 9 draws conclusions and points to future work.

3.2 Related Work

We discuss related works based on the class of problems they solve: first we describe general approaches that cover temporal joins [SSJ94; SG89] as well as temporal aggregations [BGJ06; VLSM05]; next we describe solutions for temporal joins; finally we conclude with solutions for temporal aggregations. Temporal anti-joins have received very little attention: only temporal alignment [DBG12] offers a solution for computing them.

General solutions: Dignös et al. [DBG12] proposed an approach that computes temporal operators by first producing all time intervals that appear in the result (through a normalization or alignment operation [ABPT01]), and then applies the corresponding non-temporal operator to relations with adjusted time intervals. The interval adjustment is computed with a left outer join on T with inequality conditions on start and end points of the intervals. This is a difficult to optimize primitive and is computed through a nested-loop with a quadratic number of comparisons.

The Timeline Index [KMV$^+$13] has been introduced to compute temporal joins and temporal aggregations with the main memory system SAP HANA. The Timeline Index consists of a *Version Map* that stores an Event ID for each T_s and T_e, and of an *Event List* that stores, for each Event ID, the ID of the tuples starting (indicated by 1) and ending (indicated by 0) their validity. Since the index tracks all tuples that are valid at each time point, temporal queries can be implemented by scanning Event List and Version Map concurrently. Temporal aggregates are computed cumulatively while scanning the index. For COUNT, the index is scanned and, for each interval delimited by two timestamps, the count is incremented or decremented according to the number of 0s and 1s. For SUM and AVG each timestamp requires a look-up to fetch the value of the tuple(s) originating or ending and incrementally update the aggregate value. For MIN or MAX, while scanning the index, a list of the Top-K Min/Max values is kept (to use in case the current Min/Max value ceases its validity). For each newly fetched tuple, the validity of each of the K tuples must be checked. No solution is given for determining K. Temporal Joins are computed using sort-merge on the indexes. After a joined pair is build, a look-up for each tuple ID is done

(implying that, if a tuple is a join match for k tuples in $\boldsymbol{R}$, k look-ups for the same tuple are done). This method inherits the disadvantages of traditional index joins, i.e., it is only efficient when few index look-ups are done. We experimentally show that this approach does not scale when the number of tuples (and look-ups) grows, and it deteriorates if the data does not fit into memory.

Solutions for joins: Dignös et al. [DBG14] introduced Overlap Interval Partitioning (*OIP*). The approach divides the time domain into k granules, creates partitions with increasing length that span the entire time domain, and puts each tuple into the shortest partition into which the tuple fits. The join is computed by identifying for each outer partition the overlapping inner partitions. Finding the overlapping partitions is very efficient, but a nested-loop is necessary to join partitions with overlapping tuples. This is a performance bottleneck, and when joining partitions with shorts intervals many unproductive comparisons happen since short tuples overlap with only few other tuples. If the length of the data history increases, the number of short partitions increases too, causing a high number of unproductive comparisons.

Enderle et al. [EHS04] proposed the Relational Interval Tree [KPS00] to compute temporal joins. This approach is index-based, similar to the TimeLine Index, but can be applied to joins only. As mentioned above, index-based techniques are good for few look-ups but, even if a single look-up is fast, cannot compete with more advanced techniques for computing joins if the number of index look-ups is high.

A Sweepline algorithm has been proposed by Arge et al. [APR$^+$98]. It sorts the relations by T_s, and, while scanning the relations, keeps a list of the active $\boldsymbol{R}$ (and $\boldsymbol{S}$) tuples. When a new $\boldsymbol{R}$ ($\boldsymbol{S}$) tuple is fetched, it is compared with all active $\boldsymbol{S}$ ($\boldsymbol{R}$) tuples. If an active tuple ceases its validity, it is removed from the list. The allocation and deallocation yields a poor memory locality, since after a series of insertions and deletions into the list of active tuples, the elements of the list become scattered in memory [PHD16]. This causes random accesses when traversing the list, which are considerably slower than sequential accesses [Str12]. Piatov et al. [PHD16] address this drawback by pre-allocating the space for the active tuples and, when an active tuple is removed from the list, the last inserted active tuple is moved to the free place. This requires that all tuples of the relation have the same size, which is not a realistic assumption in the general case.

MapReduce [CGN$^+$14] has been used to compute interval joins. The proposed approach partitions the time domain into q segments, and assigns to each reducer $\boldsymbol{R}_i$ all tuples overlapping

the i-th segment. Similar to other approaches it uses a nested-loop to join the tuples of two partitions, outputs the joined tuples, and broadcasts the tuples that span multiple segments to the other reducers. A similar approach that is not MapReduce-based has been proposed by Soo et al. [SSJ94]. Both approaches do not give an efficient solution for the nested-loop join between partitions.

Solutions for aggregations: In order to incrementally compute temporal aggregates, the Aggregation Tree has been proposed [KS95]. The approach has two limitations. First, the entire tree must be kept in memory. For a relation $\boldsymbol{R}$, the size of the tree is up to $2n$ (i.e., the number of different values for T_s and T_e). Second, if the input is sorted by T_s (as is often the case for temporal data), the aggregation tree will be unbalanced, and the time to create it is $O(n^2)$. The Balanced Aggregate Tree [MFVLI03] addresses the unbalancedness of the Aggregation Tree with a red-black tree. Since the tree stores time instants rather than time intervals it cannot be used to compute Min/Max aggregations. Moreover, to determine an aggregate value at a specific point in time, the tree must be scanned from the beginning to the look-up time point. The SB-Tree [YW03] reduces the number of tree nodes since multiple intervals are stored in each node (like a B-Tree), each with its corresponding aggregate value. All approaches can only be applied to distributive aggregation functions [GCB$^+$97] and must duplicate the index for each aggregation function. Our partitioning is run once and also works for non-distributive functions (e.g., Standard Deviation).

Moon et al. [MFVLI03] present a scalable algorithm based on buckets. They partition the time domain into q uniform buckets and assign to each bucket every tuple that overlaps. Tuples spanning multiple buckets are split and assigned to each overlapping bucket. Aggregation is applied inside each bucket by using one of the above mentioned algorithms. To reconstruct the tuples that have been split, adjacent result tuples are merged if they have the same aggregation value. This violates change preservation (lineage) [BJS00] because if two adjacent result tuples have the same aggregation value but originate from different tuples, the result will only include one tuple instead of two.

Sort-aggregate [Gra93] is a common technique to compute non-temporal aggregates based on sorting and is implemented in many commercial DBMSs. It sorts the data by the grouping attributes, and then computes the aggregate over the tuples within the same group (which, after the sorting, are placed next to each other). This approach can also be applied to temporal data (e.g., sorting the relation by T_s), but backtracking is needed to fetch tuples that have been scanned

before but are still valid. As for sort-merge, we experimentally show that this approach becomes quadratic as soon as one tuple with a long time interval exists.

3.3 Preliminaries

3.3.1 Notation

We assume a relational schema $(T, R_1, \ldots, R_m)$ where $R_1, \ldots, R_m$ are the non-temporal attributes, and $T = [T_s, T_e)$ is an interval attribute with T_s and T_e being, respectively, its inclusive starting and exclusive ending points. $\boldsymbol{R}$ is a relation over schema $(T, R_1, \ldots, R_m)$ with cardinality n. For a tuple $r \in \boldsymbol{R}$ and an attribute R_i, $r.R_i$ denotes the value of R_i. Given tuples r and s, r is *disjoint* from s iff $r.T_e \leq s.T_s \vee r.T_s \geq s.T_e$, otherwise the tuples are *overlapping*. For example, the tuples $([1,3), a)$ and $([2,6), b)$ are overlapping, while the tuples $([1,3), a)$ and $([8,9), c)$ are disjoint.

Table 3.1 summarizes the symbols and notation that we use in this Chapter. We use subscripts to

Symbol	Meaning
$\boldsymbol{R}$	Relation
$\boldsymbol{R}_i$	i-th $\mathcal{DIP}$ partition of $\boldsymbol{R}$
n	# of tuples of a relation
c	# of $\mathcal{DIP}$ partitions of a relation
b	size of partition in # of blocks
B	size of relation in # of blocks
$r.X$	lead of tuple r

Table 3.1: Notation.

refer to specific relations. For example, $c_{\boldsymbol{R}}$ denotes the number of $\mathcal{DIP}$ partitions of relation $\boldsymbol{R}$, while $c_{\boldsymbol{S}}$ denotes the number of $\mathcal{DIP}$ partitions of relation $\boldsymbol{S}$.

3.3.2 Temporal Operators

Table 3.2 lists and defines a temporal join, a temporal anti-join, and a temporal aggregation. As usual, the semantics of a temporal operator are defined by snapshot reducibility [Sno95] and change preservation [DBG12; BJS00]. Briefly, *snapshot reducibility* ensures that the result of a

Oper.	Definition
$\boldsymbol{R} \bowtie_T \boldsymbol{S}$	$\{<\tau, r.R_1, \ldots, r.R_l, s.S_1, \ldots, r.R_m> \mid$ $r \in \boldsymbol{R} \wedge s \in \boldsymbol{S} \wedge$ $overlap(r,s) \wedge \tau = (r.T \cap s.T)\}$
$\boldsymbol{R} \triangleright_T \boldsymbol{S}$	$\{<\tau, r.R_1, \ldots, r.R_l> \mid$ $r \in \boldsymbol{R} \wedge \tau.T_s \geq r.T_s \wedge \tau.T_e \leq r.T_e \wedge$ $\nexists s \in \boldsymbol{S}\big(overlap(s,\tau)\big) \wedge$ $\big(\tau.T_s = r.T_s \vee \exists u \in \boldsymbol{S}(\tau.T_s = u.T_e)\big) \wedge$ $\big(\tau.T_e = r.T_e \vee \exists v \in \boldsymbol{S}(\tau.T_e = v.T_s)\big)\}$
$\vartheta^T_{f(A)}\boldsymbol{R}$	$\{<\tau, f(\boldsymbol{R}'.A)> \mid$ $r, s \in \boldsymbol{R} \wedge len(\tau) > 0 \wedge$ $\tau.t_s = (r.T_s \vee r.T_e) \wedge \tau.t_e = (s.T_s \vee s.T_e) \wedge$ $\forall u \in \boldsymbol{R}\big(overlap(u,\tau) \leftrightarrow (\tau - u.T = \emptyset \wedge u \in \boldsymbol{R}')\big)\}$

Table 3.2: Semantics of temporal operators.

temporal operator at any time point p is equal to the result of the corresponding non-temporal temporal operator applied to the input that is valid at p. Thus, the time interval τ of the output tuples depends on the semantics of the temporal operator. For a join, it is the time during which the outer and inner tuples are both valid; for an anti-join, it is the time during which an outer tuple is valid and no inner tuple is; for an aggregation, it is the largest interval during which a set of tuples is valid. *Change preservation* ensures that the result of a temporal operator respects lineage. Thus, any change in the input tuples is reflected in the intervals of the output tuples. For example, in Figure 3.16, even if they share the same aggregation value, two output tuples are returned between [8,10) and [10,11) since the tuples valid during [8,10) are different from the tuples valid during [10,11).

The result of the operators applied to our running example are shown in Figure 3.10 for a join, Figure 3.12 for an anti-join, and Figure 3.16 for an aggregation, and will be explained in Section 3.6.

3.3.3 Sort-Merge over Interval Data

This section shows that, with just one long-lived tuple, the number of unproductive comparisons for sort-merge and sort-aggregate over relations with overlapping tuples gets quadratic.

Let $\boldsymbol{L} = \{L_1, \ldots, L_l\}$ be the longest intervals in a relation for all possible points of the time domain. In Figure 3.5, we have $\boldsymbol{L} = \{L_1, L_2, L_3\}$ (for simplicity we assume that L_1, L_2, L_3 do not overlap).

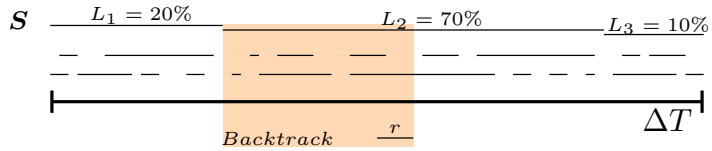
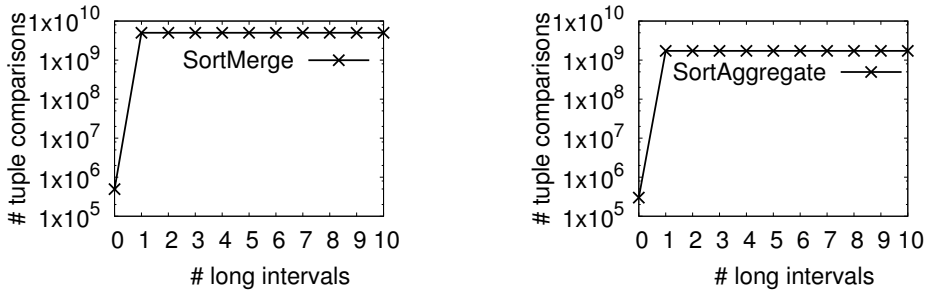


Figure 3.5: For $r \in \boldsymbol{R}$, on average, half of the tuples within L_i are compared because of backtracking.

The cost of sort-merge in terms of tuple comparisons is $Cost$(Merge) + $Cost$(Backtracking). $Cost$(Merge) $= n + n - 1$ is the cost for scanning the two input relations, and $Cost$(Backtracking) $= n \times \frac{\bar{\boldsymbol{L}}}{|\Delta T|} \times \frac{n}{2}$ quantifies the number of tuples rescanned. In this cost $|\Delta T|$ is the length of the time domain, and $\bar{\boldsymbol{L}} = \left(\sum_{i=1}^{l} L_i^2\right) / \left(\sum_{i=1}^{l} L_i\right)$ is the *weighted* average length of L_i in $\boldsymbol{S}$. $\bar{\boldsymbol{L}}$ is weighted since the number of $\boldsymbol{R}$ tuples that fall within L_i is proportional to its length. For example, in Figure 3.5, assuming uniform distribution, most of the $\boldsymbol{R}$ tuples will be within L_2, whose duration is 70% of the time domain. $\frac{\bar{\boldsymbol{L}}}{|\Delta T|} \times n$ quantifies the number of tuples within L_i. On average r ends in the middle of L_i, which means that for each $r \in \boldsymbol{R}$, half of the tuples within L_i are refetched in $\boldsymbol{S}$ because of backtracking. Even if only *one* interval L_1 spans the entire time domain, then $\boldsymbol{L} = \{L_1\}$ and $\frac{\bar{\boldsymbol{L}}}{|\Delta T|} = 1$, and the number of comparisons becomes quadratic. This is shown in Figure 3.6(a), where a temporal join on the fact table of the Swiss Feed Data Warehouse [TBBC12] is computed. We show that as soon as measurements that are time invariant are taken into account (e.g., the *Protein Digestibility* value), sort-merge becomes inefficient.



(a) Sort-merge (b) Sort-aggregate

Figure 3.6: Sort-merge and sort-aggregate deteriorate as soon as a single long interval exists.

Lemma 5. *(Upper-bound of sort-merge) The number of unproductive comparisons for a temporal join $\boldsymbol{R} \bowtie_T \boldsymbol{S}$ using sort-merge is upper-bounded by n^2, where n is the number of tuples of $\boldsymbol{R}$ and $\boldsymbol{S}$.*

Proof. If $\boldsymbol{S}$ includes an interval L_1 that spans the entire time domain, then $\frac{\bar{L}}{|\Delta T|} = 1$. If each $r \in \boldsymbol{R}$ overlaps L_1 and $r.T_e > L_1.T_e$, backtracking must refetch all $\boldsymbol{S}$ tuples. In the worst case, each $r \in \boldsymbol{R}$ only overlaps with L_1, and we get: $Cost_{SM}(\boldsymbol{R} \bowtie_T \boldsymbol{S}) - |\boldsymbol{R} \bowtie_T \boldsymbol{S}| = n \times n - n = O(n^2)$ unproductive comparisons. □

Figure 3.6(b) shows that also sort-aggregate [Gra93], i.e., temporal aggregation computed using sorting, suffers from a quadratic number of unproductive comparisons. For non-temporal data sort-aggregate makes only one scan to compute the aggregate because, after the sorting, all tuples of the same group are consecutive. When dealing with time intervals, sort-aggregate must backtrack to fetch tuples that have been scanned before but are still valid. This yields unproductive comparisons.

3.4 Disjoint Interval Partitioning

Definition 2. *($\mathcal{DIP}$ partition).* Consider a relation $\boldsymbol{R}$ with schema $(T, R_1, \ldots, R_m)$. A $\mathcal{DIP}$ partition $\boldsymbol{R}_i \subseteq \boldsymbol{R}$ is a subset of $\boldsymbol{R}$ such that:

$$\forall (r, s) \in \boldsymbol{R}_i \big(r \neq s \Rightarrow disjoint(r, s) \big)$$

Thus, a $\mathcal{DIP}$ partition $\boldsymbol{R}_i$ is a set of non-overlapping tuples from $\boldsymbol{R}$. In Figure 3.3 we have three outer $\mathcal{DIP}$ partitions ($\boldsymbol{R}_1, \boldsymbol{R}_2, \boldsymbol{R}_3$) and two inner $\mathcal{DIP}$ partitions ($\boldsymbol{S}_1, \boldsymbol{S}_2$). Tuples of different partitions may overlap, but within a single partition all tuples are disjoint.

3.4.1 Efficient Merging of $\mathcal{DIP}$ Partitions

We use $\mathcal{DIP}$ to speed up the *merge* in sort-merge computations. The advantage of $\mathcal{DIP}$ is that a temporal operator can be computed between the $\mathcal{DIP}$ partitions using a merge procedure that does not have to backtrack, i.e., does only scan of the partitions with sequential IOs only.

Lemma 6. *(No Backtracking) Consider two $\mathcal{DIP}$ partitions $\boldsymbol{R}_i$ and $\boldsymbol{S}_j$. During a sort-merge computation no backtracking must be done in $\boldsymbol{S}_j$ to find the tuples that overlap $r \in \boldsymbol{R}_i$.*

Proof. Let $r_1, r_2 \in \boldsymbol{R}_i$ such that $r_1.T_e \leq r_2.T_s$. Since the partitions are sorted (e.g., by T_s), r_1 always precedes r_2. Tuples r_1 and r_2 are disjoint since they are in the same $\mathcal{DIP}$ partition. Since all tuples in $\boldsymbol{S}_j$ are disjoint, at most one tuple s_k may exists that overlaps r_1 such that $s_k.T_e > r_1.T_e$ (cf. Figure 3.7). All tuples that precede s_k end before r_2 starts and therefore cannot overlap r_2. □

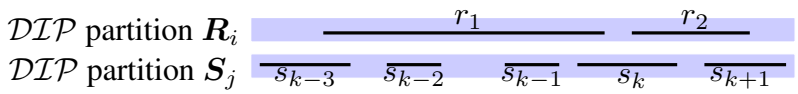


Figure 3.7: Illustration of Lemma 6.

Lemma 6 guarantees that, if s_k is the last tuple that overlaps r, there is no need to rescan any tuple before s_k to find the matches for the next $\boldsymbol{R}_i$ tuple. This means that a merge procedure between $\mathcal{DIP}$ partitions can be computed without backtracking, i.e., by just making one scan of the input partitions.

Figure 3.8 illustrates that Lemma 6 also holds when multiple outer partitions are merged simultaneously with $\boldsymbol{S}_j$. The scan of an outer partition (e.g., $\boldsymbol{R}_1$) proceeds until all tuples overlapping s_k are found. Since in $\boldsymbol{R}_1$ only the last scanned tuple can overlap with s_{k+1}, we mark its position in $\boldsymbol{R}_1$, and process $\boldsymbol{R}_2$ (and later $\boldsymbol{R}_3$) to find the other join matches of s_k. After all join matches have been found, s_{k+1} is fetched and the scan of the outer partitions resumes from the previously marked positions. Of the tuples previously accessed in $\boldsymbol{R}_1, \boldsymbol{R}_2$ and $\boldsymbol{R}_3$, only the marked ones (i.e., the bold-faced ones in Figure 3.8) can overlap with s_{k+1}. Section 3.7 describes the implementation of $\mathcal{DIP}Merge(\{\boldsymbol{R}_1, \ldots, \boldsymbol{R}_m\}, \boldsymbol{S}_j)$, i.e., a Merge procedure between m outer and one inner $\mathcal{DIP}$ partition.

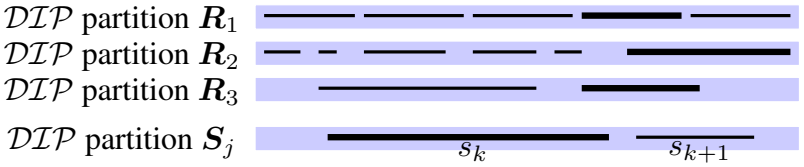


Figure 3.8: Efficient $\mathcal{DIP}Merge$ without backtracking: in each outer partition, no tuple before the one scanned last for s_k can overlap s_{k+1}.

3.5 Efficient Data Partitioning

We use $\mathcal{DIP}$ as the essential first step to efficiently compute temporal operators. Since the number of unproductive comparisons is limited by the number of $\mathcal{DIP}$ partitions, we first provide the $CreateDIP$ algorithm to partition the input relation into the *minimum* number of $\mathcal{DIP}$ partitions. $CreateDIP$ outputs the partitions with their elements already sorted. Thus, before computing a *$\mathcal{DIP}$Merge* between $\mathcal{DIP}$ partitions no additional sorting is required. In the experiments, we show that this makes the runtime for computing temporal operators very small.

3.5.1 The Partitioning Algorithm $CreateDIP$

Algorithm $CreateDIP(\boldsymbol{R})$ sorts the input relation $\boldsymbol{R}$ by $T_s, (T_e - T_s)_{desc}$, i.e., by T_s and by the interval length starting from the longest. It then scans the data and places blocks of non-overlapping tuples in the partitions. In order to determine if the next tuple t overlaps with the tuples in the current $\mathcal{DIP}$ partition $\boldsymbol{R}_i$, the following Lemma asserts that it is enough to compare t with only the last tuple of $\boldsymbol{R}_i$ since the data is sorted.

Lemma 7. *(Transitivity) Consider tuples $r, s \in \boldsymbol{R}_i$ and a new tuple t such that $r.T_s \leq s.T_s \leq t.T_s$. Then $disjoint(r,s) \wedge disjoint(s,t) \iff disjoint(r,t)$.*

Proof. The end point of an interval is always larger than the start point: $s.T_e > s.T_s$. Since the tuples in a $\mathcal{DIP}$ partition do not overlap, we have $r.T_e < s.T_s$. Since $s.T_s \leq t.T_s$ (recall that we process tuples ordered by T_s), $r.T_e \leq t.T_s$ follows: r does not overlap t. □

There are various ways partitions can be filled. Current approaches strive to balance the size of the partitions, so that the cost of the nested-loop between two partitions is optimized (e.g., for two relations with $10k$ tuples and two partitions each, if the partitions have size $5k$, then the two nested-loops perform $5k \times 5k + 5k \times 5k = 50M$ comparisons; if they have size $9k$ and $1k$, then they perform $9k \times 9k + 1k \times 1k = 82M$ comparisons). The performance of our approach is independent of how the tuples are distributed: we just must guarantee that the number of $\mathcal{DIP}$ partitions is minimal.

We start by creating the first partition $\boldsymbol{R}_1$ in the partition list $\mathcal{L}$, i.e., $\mathcal{L} = <\boldsymbol{R}_1>$, and populate $\boldsymbol{R}_1$ until we encounter an overlapping tuple. At this point, we prepend a new partition to the partition list (i.e., $\mathcal{L} = <\boldsymbol{R}_2, \boldsymbol{R}_1>$). Again, we put the overlapping tuple into $\boldsymbol{R}_2$, together with

all following tuples until we encounter an overlapping tuple r. At this point, before creating a new partition, we first try to place r in one of the other partitions (e.g., $\boldsymbol{R}_1$). In order to not visit more than one partition unsuccessfully, we reposition the last written partition (i.e., $\boldsymbol{R}_2$) in the partition list, so that the partitions in the partition list are sorted according to the T_e of their last tuple. This guarantees that the first partition of the list is the one ending before all others: if r does not overlap with it, we place it in there, otherwise we know that r also overlaps with all other partitions, and a new one must be created. There is no need to visit any partition other than the first one of the list.

Algorithm 5: $CreateDIP(\boldsymbol{R})$

```
1  Sort(R) by T_s, (T_e − T_s)_desc ;
2  R_1 ← ∅ ;
3  L = <R_1> ;
4  while (r = fetchtuple(R)) ≠ null do
5      if L.head = ∅ ∨ disjoint(L.head.last, r) then
6          Add r to L.head ;
7      else
8          l = L.head ;
9          while l.next ≠ null ∧ l.last.T_e > l.next.last.T_e do
10             Swap(l, l.next) ;
11         if overlap(L.head.last, r) then
12             R_{len(L)+1} = ∅;
13             L = prepend(R_{len(L)+1}, L) ;
14         Add r to L.head ;
15 return L;
```

Algorithm 5 describes the details of our implementation. $\mathcal{L}$ is the partition list ($\mathcal{L}$ is implemented as a list of pointers that point to the partitions). For each $r \in \boldsymbol{R}$, if r does not overlap with the last tuple of the first partition of the list (i.e, $\mathcal{L}.head.last$), we add r to $\mathcal{L}.head$ (lines 5-6). Otherwise, in lines 9-10, we first swap $\mathcal{L}.head$ with each following partition in $\mathcal{L}$ that ends before it (this step just updates pointers). As a result, $\mathcal{L}$ is sorted according to the T_e value of the tuples last inserted into the partitions. As illustrated in Figure 3.9, if r overlaps with $\mathcal{L}.head.last$, then it also overlaps with all other partitions. Therefore, a new empty partition (i.e, $\boldsymbol{R}_{len(\mathcal{L})+1}$) can be created without checking any partition other than $\mathcal{L}.head$ (lines 11-13). We finally add r to $\mathcal{L}.head$ (line 14). When all tuples have been processed, the algorithm returns a partition list with the minimum number of $\mathcal{DIP}$ partitions.

3.5.2 Long Lived Tuples

In contrast to other approaches, $CreateDIP$ stays robust if the data includes long lived tuples (e.g., tuples that span a large portion of the time domain). Remember that the input tuples are sorted by T_s. If a long lived tuple is put into a partition $\boldsymbol{R}_i$, most of the following tuples will overlap with it, and they will never be placed in $\boldsymbol{R}_i$. $CreateDIP$ does not continuously try to populate a partition with a long lived tuple unsuccessfully. If a new tuple overlaps with the long lived tuple in $\boldsymbol{R}_i$, $CreateDIP$ pushes $\boldsymbol{R}_i$ to the end of the partition list (lines 9-10) since the T_e of the long lived tuple is large. $\boldsymbol{R}_i$ will not be visited anymore until tuples with a larger T_e value have been put in all the other partitions.

Note that our approach is general and does not modify long lived tuples. In contrast, approaches that split long lived tuples into pieces violate change preservation (and might deteriorate if many long lived tuples exist).

3.5.3 Correctness and Optimality

We prove that $CreateDIP$ returns the minimum number of partitions with disjoint tuples. The *minimum number c of DIP partitions* is given by the maximum number of tuples that are valid at some time point. In the example in Figure 3.9 this is the case at time τ when 5 tuples are valid.

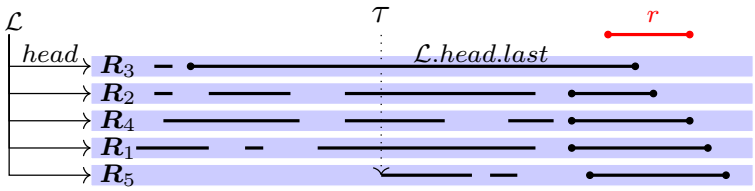


Figure 3.9: The number of partitions for $\mathcal{DIP}$ is given by the maximal number of tuples that overlap with each other in a relation. This is the case for time τ. List $\mathcal{L}$ is sorted by the T_e of the last tuple of the partitions: if the next tuple r overlaps with $\mathcal{L}.head$, then it overlaps with all the other partitions, and a new partition is needed.

Lemma 8. *(Correctness) A partition produced by CreateDIP is a $\mathcal{DIP}$ partition.*

Proof. We must show that all tuples in a partition are disjoint. The algorithm sorts the input data by T_s and populates the partition $\mathcal{L}.head$ as long as the current $\boldsymbol{R}$ tuple (i.e., r) does not overlap with the tuples of $\mathcal{L}.head$. In order to determine if r overlaps with the tuples of $\mathcal{L}.head$, it is

sufficient (cf. Lemma 7) that r does not overlap with the last element of $\mathcal{L}.head$. If r overlaps with all partitions, $CreateDIP$ creates a new empty partition at $\mathcal{L}.head$ into which tuple r is inserted. □

Lemma 9. *(Optimality) If c is the largest number of tuples that overlap contemporaneously in relation $\boldsymbol{R}$, CreateDIP returns c DIP partitions.*

Proof. Given the minimum number c of partitions, we show that $CreateDIP$ does not create more or less partitions.

1. If $CreateDIP$ returned less than c partitions, then, at a certain point, two overlapping tuples must be placed in the same partition, i.e., in $\mathcal{L}.head$. $CreateDIP$ inserts a tuple r into $\mathcal{L}.head$ only if it does not overlap with its last element, and Lemma 7 guarantees that in this case r does not overlap with any other tuple in $\mathcal{L}.head$. Since two overlapping elements cannot be in the same partition, it follows that our algorithm cannot return less than c partitions.

2. If $CreateDIP$ returned more partitions than c, say $c+1$, then there exists a partition $\boldsymbol{R}_i$ such that $\forall r\big(r \in \boldsymbol{R}_i \Rightarrow disjoint(r,t)\big)$, where t is the first tuple of $\boldsymbol{R}_{c+1}$ (i.e., the element that was first placed in this partition). However, before creating $\boldsymbol{R}_{c+1}$, $CreateDIP$ ensures that $\mathcal{L}.head$ is the partition whose last tuple ends earliest (lines 8-10) and tries to put t in it. If there exists a partition $\boldsymbol{R}_i$ not overlapping t, then $\boldsymbol{R}_i$ is moved at the beginning of $\mathcal{L}$. After this, since $\mathcal{L}.head$ does not overlap with t, the condition on line 11 of $CreateDIP$ is false, and $\boldsymbol{R}_{c+1}$ will not be created.

□

3.5.4 Cost of $CreateDIP$

The runtime complexity of our partitioning is given by the sum of: *i)* the cost of sorting, i.e., $O(n \log n)$, and *ii)* the cost of the algorithm itself, i.e., $O(c \times n)$, since each of the n tuple is compared against one partition and, for each tuple, at most $c-1$ partition switches are performed. For data with a long history, i.e., data collected over many years, c is small compared to the size of the relation. For example, in a database storing the bookings of a hotel, c is equal to the number of rooms (e.g., all rooms are occupied on a given day), which is smaller than all bookings recorded

since the beginning. In data collected in a network of sensors, c is the number of sensors (e.g., all sensors record a value at the same time). Also, note that $c \times n$ is the upper-bound for *CreateDIP* since in the average case, when an overlapping tuple is fetched, $\mathcal{L}.head$ is not usually pushed all the way to the end of the list.

3.6 Temporal Operators

Our approach reduces a temporal operator O_T over an entire relation to a sequence of temporal operators over $\mathcal{DIP}$ partitions, i.e., $O_T^{\mathcal{DIP}}$. We apply our findings to the computation of temporal joins, anti-joins, and aggregations, and show that each $O_T^{\mathcal{DIP}}$ can be implemented with a merge procedure that does not backtrack.

3.6.1 Temporal Join

A temporal join $\boldsymbol{R} \bowtie_T \boldsymbol{S}$ returns the pairs (r, s), with $r \in \boldsymbol{R}$ and $s \in \boldsymbol{S}$, whose time interval T overlaps. Figure 3.10 illustrates the join result of our example relations. It computes the price difference between the luxury suites of hotel $\boldsymbol{R}$ and those of hotel $\boldsymbol{S}$. For example, the second output row says that suite #1 of hotel $\boldsymbol{R}$ and suite #2 of hotel $\boldsymbol{S}$ have a price difference of 10 $ during time $[1, 2)$, while the third row says that, at time $[3, 4)$, they cost the same.

$\Pi_{T,\boldsymbol{R}.\#,\boldsymbol{S}.\#,\boldsymbol{R}.\$-\boldsymbol{S}.\$}(\boldsymbol{R} \bowtie_T \boldsymbol{S})$

T	$\boldsymbol{R}.\#$	$\boldsymbol{S}.\#$	$\boldsymbol{R}.\$ - \boldsymbol{S}.\$$
$[1, 5)$	1	6	20
$[1, 2)$	1	2	10
$[3, 4)$	1	2	0
..	..	..	..
$[10, 11)$	2	2	-20
$[10, 11)$	5	2	-10

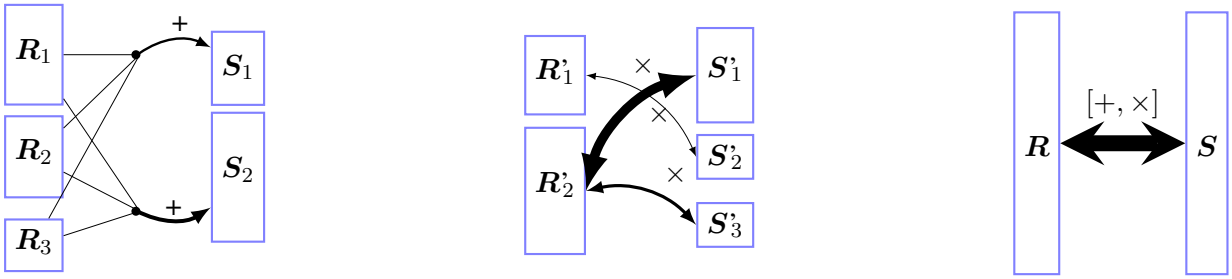
Figure 3.10: Temporal join applied to the running example.

To compute a temporal join using $\mathcal{DIP}$, sets of m outer partitions (e.g., $\{\boldsymbol{R}_1, \ldots, \boldsymbol{R}_m\}$) are joined with each inner partition until all outer partitions have been processed:

$$\boldsymbol{R} \bowtie_T \boldsymbol{S} \iff \bigcup_{i=1}^{c_{\boldsymbol{R}}/m} \bigcup_{j=1}^{c_{\boldsymbol{S}}} \left(\{\boldsymbol{R}_{i*m-m+1}, \ldots, \boldsymbol{R}_{i*m}\} \bowtie_T^{\mathcal{DIP}} \boldsymbol{S}_j\right) \tag{3.1}$$

Thus, to compute a temporal join, we compute $\frac{c_{\boldsymbol{R}} \times c_{\boldsymbol{S}}}{m}$ $\mathcal{DIP}$*Merge*s. Each $\mathcal{DIP}$*Merge* joins m outer partitions with each inner partition: first $\{\boldsymbol{R}_1, \ldots, \boldsymbol{R}_m\}$ are joined with $\boldsymbol{S}_1$, then with $\boldsymbol{S}_2$, etc.

Figure 3.11 illustrates the differences between $\mathcal{DIP}$ and other approaches on our running example. The thickness of the arrows is proportional to the cost of each join in terms of number of comparisons. With $\mathcal{DIP}$ many outer partitions can be processed simultaneously. Furthermore, even if the total number of merges between partitions might be higher for $\mathcal{DIP}$, the cost of each $\mathcal{DIP}$*Merge* is small compared to the cost of the other approaches since it requires only one scan of the input partitions (it is computed in linear rather than quadratic time).



(a) $\mathcal{DIP}$ (b) *OIP* (c) SM

Figure 3.11: $\mathcal{DIP}$ joins with linear cost (indicated by a + sign) many outer partitions with each inner partition. *OIP* joins with quadratic cost (indicated by a $\times$ sign) each outer partition with few inner partitions. Sort-merge backtracks without any guarantee about the complexity (i.e., ranges from linear to quadratic).

Example 9. We use Equation (1) to compute a temporal join between the relations of our running example, i.e., $\boldsymbol{R}$ with $c_{\boldsymbol{R}} = 3$ and $\boldsymbol{S}$ with $c_{\boldsymbol{S}} = 2$. Example 2 illustrates the process. Clearly, the join of all $\mathcal{DIP}$ partitions is done with much less unproductive comparisons than the sort-merge join in Figure 3.2.

Equation (1) shows that the higher m, the fewer $\mathcal{DIP}Merge$s are computed. The value of m is given by the number of partitions that can be processed simultaneously. In typical commercial operating systems this is about 10^4 (the number of files a process is allowed to keep open at a time). We will show that in a worst case scenario, i.e., when *all* tuples overlap and n partitions are created, m is the factor by which we reduce the quadratic worst case I/O cost for computing a temporal join, which is significant.

CPU Cost

We quantify the CPU overhead in terms of *unproductive comparisons*, i.e., the number of tuples comparisons that do not produce an output tuple. We determine an upper-bound for the number of unproductive comparisons. For simplicity, we use c to indicate both the number of partitions of $\boldsymbol{R}$ and those of $\boldsymbol{S}$, and $\frac{n}{c}$ to indicate the number of tuples of a partition.

Lemma 10. *Consider relations $\boldsymbol{R}$ and $\boldsymbol{S}$ that have been partitioned into c $\mathcal{DIP}$ partitions each. The number of unproductive comparisons for computing $\boldsymbol{R} \bowtie_T \boldsymbol{S}$ using $\mathcal{DIP}$ is upper-bounded by $c \times n$.*

Proof. From Equation (3.1) we get

$$\text{CPU}(\boldsymbol{R} \bowtie_T \boldsymbol{S}) = \text{CPU}\Big(\bigcup_{i=1}^{c/m} \bigcup_{j=1}^{c} (\{\boldsymbol{R}_{i*m-m+1}, \ldots, \boldsymbol{R}_{i*m}\} \bowtie_T^{\mathcal{DIP}} \boldsymbol{S}_j)\Big)$$

Operator $\bowtie_T^{\mathcal{DIP}}$ is implemented as a $\mathcal{DIP}Merge$, i.e., a procedure that in each iteration either advances one (outer or inner) tuple or switches the current outer partition (see our implementation of $\mathcal{DIP}Merge$ in Algorithm 2). Thus, the number of iterations is given by the total size of the m outer partitions, plus $m-1$ partition switches per inner tuple, plus the size of the the inner partition. We get:

$$\begin{aligned}
\text{CPU}(\boldsymbol{R} \bowtie_T \boldsymbol{S}) &= \sum_{i=1}^{c/m} \sum_{j=1}^{c} (\tfrac{n}{c} * m + (m-1)\tfrac{n}{c} + \tfrac{n}{c}) \\
&= \sum_{i=1}^{c/m} (n * m + n * m) \\
&\approx c * n
\end{aligned}$$

The number of unproductive comparisons is $CPU(\boldsymbol{R} \bowtie_T \boldsymbol{S}) - |\boldsymbol{R} \bowtie_T \boldsymbol{S}|$, i.e., the number of comparisons minus the number of result tuples. In the worst case we have 0 result tuples, and we get $c * n$ unproductive comparisons. □

I/O Cost

This section quantifies the number of block I/Os for computing a temporal join using $\mathcal{DIP}$. We assume that all $\mathcal{DIP}$ partitions are equally sized, and each of them has $b = \frac{B}{c}$ blocks.

Lemma 11. *Consider relations $\boldsymbol{R}$ and $\boldsymbol{S}$ partitioned into c $\mathcal{DIP}$ partitions each. The number of I/Os for computing $\boldsymbol{R} \bowtie_T \boldsymbol{S}$ using $\mathcal{DIP}$ is min$(c, \frac{n}{m}) \times B$.*

Proof. From Equation (1) we get:

$$\text{IO}(\boldsymbol{R} \bowtie_T \boldsymbol{S}) = \text{IO}\Big(\bigcup_{i=1}^{c/m} \bigcup_{j=1}^{c} (\{\boldsymbol{R}_{i*m-m+1}, \ldots, \boldsymbol{R}_{i*m}\} \bowtie_T^{\mathcal{DIP}} \boldsymbol{S}_j)\Big)$$

With equally sized partitions we obtain:

$$= \tfrac{c}{m} \times c \times \text{IO}(\{\boldsymbol{R}_{i*m-m+1}, \ldots, \boldsymbol{R}_{i*m}\} \bowtie_T^{\mathcal{DIP}} \boldsymbol{S}_j)$$

Equation (1) shows that for $c_{\boldsymbol{S}}$ subsequent calls of $\mathcal{DIP}$*Merge* only the inner partition is changed. Since the outer partitions $\{\boldsymbol{R}_{i*m-m+1}, \ldots, \boldsymbol{R}_{i*m}\}$ are reused, we cache the first M blocks of each $\boldsymbol{R}_i$, and obtain:

$$\begin{aligned} &= \tfrac{c}{m} \times c \times \big((b - M) * m + b\big) \\ &= c^2 \times (b - M) + \tfrac{c^2}{m} \times b \end{aligned} \tag{3.2}$$

When dealing with data histories, tuples are valid at different points in time and partitions get large since old tuples do not overlap with recent ones. This means $b >> M$, and from (3.2) we get:

$$\begin{aligned} \text{IO}_{General}(\boldsymbol{R} \bowtie_T \boldsymbol{S}) &= c^2 \times b + \tfrac{c^2}{m} \times b \\ &\approx c \times B \end{aligned} \tag{3.2a}$$

where $B = c \times b$ are the blocks of an input relation. In other words, in the general case, our approach is linear in the number of partitions: independent of m it fetches each block c times.

The worst case for our approach is when $c \approx n$, i.e., many partitions exist (e.g., most tuples overlap). In such a case the partitions are small since only few non-overlapping tuples can be stored in a partition. With small partitions we have $c \approx n \iff b \leq M$, and from (3.2) we get:

$$\begin{aligned} \text{IO}_{Worst} &= c^2 \times 0 + \tfrac{c^2}{m} \times b = \tfrac{c}{m} \times B \\ &\approx \tfrac{n}{m} \times B \end{aligned} \tag{3.2b}$$

Summarizing:

$$\begin{aligned} \text{IO}(\boldsymbol{R} \bowtie_T \boldsymbol{S}) &= min(\text{IO}_{General}, \text{IO}_{Worst}) \\ &= min(c, \tfrac{n}{m}) \times B \end{aligned}$$

□

Thus, while in the general case $\mathcal{DIP}$ fetches each block c times, m helps to speed up our worst-case scenario: if m outer partitions are processed simultaneously, we reduce the number of I/Os to perform by a factor of m. This is effective already for small values of m: for example if $m = 10$ we make 10 times less I/Os. In our experiments we will show that, if m is just 0.1% the number of partitions, i.e., 0.1% of the partitions are processed simultaneously, our approach reaches the same performances as state of the art solutions that put overlapping tuples in the same partition [DBG14].

3.6.2 Temporal Anti-Join

A temporal anti-join $\boldsymbol{R} \triangleright_T \boldsymbol{S}$ returns, for each $r \in \boldsymbol{R}$, its maximal sub-intervals (if any) during which no tuple in $\boldsymbol{S}$ exists. Figure 3.12 illustrates the result of a temporal anti-join $\boldsymbol{R} \triangleright_T \boldsymbol{S}$ on our example relations. The anti-join returns the price of the luxury suites of hotel $\boldsymbol{R}$ when no suite has been booked in hotel $\boldsymbol{S}$. The result includes one tuple since $[8, 9)$ is the only time interval during which a tuple in $\boldsymbol{R}$ is valid and no tuple in $\boldsymbol{S}$ exists.

$\boldsymbol{R} \rhd_T \boldsymbol{S}$

T	#	\$
$[8, 9)$	3	75

Figure 3.12: Temporal anti-join applied to the main example

In order to take advantage of Lemma 6, and compute the anti-join without backtracking, we transform the anti-join problem into a problem of finding overlapping intervals. To do so, we do not compare $r \in \boldsymbol{R}_i$ with the time interval of $s \in \boldsymbol{S}$, but with its *lead*.

Definition 3. *(Lead).* Let s be a tuple of a relation $\boldsymbol{S}$; the lead of s, indicated by $s.X = [X_s, X_e)$, is the largest interval (if any) not overlapping any $\boldsymbol{S}$ tuple and such that $s.X_e = s.T_s$.

Example 10. In relation $\boldsymbol{Z}$ of Figure 3.13, we have $z_1.X = [-\infty, 0)$, $z_2.X = [1, 3)$, and $z_4.X = [10, 11)$. Tuple z_3 does not have a lead.

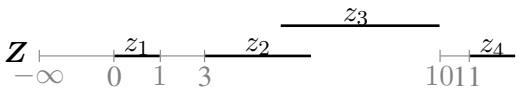


Figure 3.13: Leads of example tuples.

The lead of $s \in \boldsymbol{S}$ is the maximal interval preceding $s.T$ during which no tuple in $\boldsymbol{S}$ exists. If a tuple $r \in \boldsymbol{R}$ overlaps with $s.X$, then $r.T \cap s.X$ is the time during which r must be returned as a result tuple for $\boldsymbol{R} \rhd_T \boldsymbol{S}$. A lead has always length larger than 0. If there does not exist such an interval for s, then s does not have a lead. In a relation, e.g., $\boldsymbol{S}$, there cannot exist two leads that overlap with each other: this guarantees that no backtracking is needed for computing $\boldsymbol{R}_i \rhd_T \boldsymbol{S}$ (cf. Example 5).

The lead of a tuple $s_j \in \boldsymbol{S}$ can be computed on the fly. Since $\boldsymbol{S}$ is sorted by T_s, the lead is computed as $s_j.X = [s_{j-a}.T_e, s_j.T_s)$, with $a > 0$, where s_{j-a} is the tuple preceding s_j with the largest T_e value. If $s_j.X$ has a duration larger than 0, then s_j has a lead; otherwise it doesn't. For example, in Figure 3.15, $s_1.X$, $s_5.X$ and $s_7.X$ are the only leads. Algorithm 2 shows the details.

To compute a temporal anti-join, the first m $\mathcal{DIP}$ partitions $\{\boldsymbol{R}_1, \ldots, \boldsymbol{R}_m\}$ are anti-joined with the entire relation $\boldsymbol{S}$; the same is done for $\{\boldsymbol{R}_{m+1}, \ldots, \boldsymbol{R}_{m*2}\}$, and so on:

$$\boldsymbol{R} \rhd_T \boldsymbol{S} \iff \bigcup_{i=1}^{c_{\boldsymbol{R}}/m} (\{\boldsymbol{R}_{i*m-m+1}, \ldots, \boldsymbol{R}_{i*m}\} \rhd_T^{\mathcal{DIP}} \boldsymbol{S}) \tag{3.3}$$

Figure 3.14 illustrates that the cost for a $\mathcal{DIP}$*Merge* is linear in the size of $\{\boldsymbol{R}_1, \ldots, \boldsymbol{R}_m\}$ and $\boldsymbol{S}$.

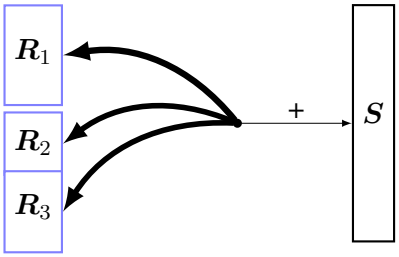


Figure 3.14: A temporal anti-join between $\boldsymbol{R}$ and $\boldsymbol{S}$ is computed by joining m outer partitions with $\boldsymbol{S}$. No backtracking is done.

Example 11. We use Equation 3.3 to compute a temporal anti-join on relations $\boldsymbol{R}$ and $\boldsymbol{S}$ of our running example (cf. Figure 3.15). Only $\boldsymbol{R}$ is partitioned. A $\mathcal{DIP}$*Merge* is applied between $\{\boldsymbol{R}_1, \boldsymbol{R}_2, \boldsymbol{R}_3\}$ and $\boldsymbol{S}$, without any backtracking. Tuples r_1 and s_1 are the first to be fetched, and $s_1.X = [-\infty, s_1.T_s) = [-\infty, 0)$. Tuple r_1 does not overlap with $s_1.X$. Since $s_1.X$ ends before r_1, we switch to $\boldsymbol{R}_2$ and r_3 is fetched. Tuple r_3 does not overlap with $s_1.X$, and, r_4 is fetched from $\boldsymbol{R}_3$. We can conclude that $s_1.X$ does not overlap with any outer tuple, therefore a new tuple is fetched from $\boldsymbol{S}$, i.e., s_2. Since the length of $s_2.X = [8, 1)$ is not larger than 0 (i.e., s_2 does not have a lead), no output is produced for r_1, nor for r_3, nor for r_4. Eventually s_5 is fetched, whose lead is larger than 0. Since r_4 overlaps with $s_5.X$, a result tuple for r_4 with time $r_4.T \cap s_5.X = [8, 9)$ is produced.

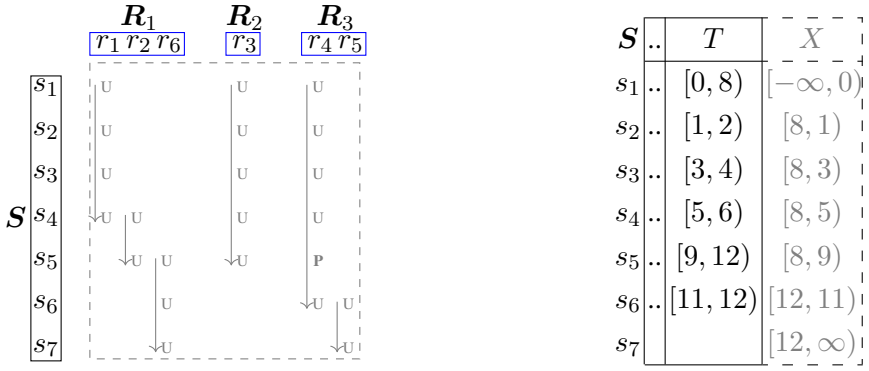


Figure 3.15: Anti-join computed using $\mathcal{DIP}$: for each $\boldsymbol{R}_i$ tuple, its timestamp is compared with the lead $s.X$ during which no tuple exists in $\boldsymbol{S}$; no backtracking is needed.

CPU Cost

We determine the CPU cost as the upper-bound for the number of unproductive comparisons for a temporal anti-join. Again, we use c to indicate the number of partitions.

Lemma 12. *Consider relations $\boldsymbol{R}$ and $\boldsymbol{S}$, and let c be the number of $\mathcal{DIP}$ partitions of $\boldsymbol{R}$. The number of unproductive comparisons for computing $\boldsymbol{R} \triangleright_T \boldsymbol{S}$ using $\mathcal{DIP}$ is upper-bounded by $c \times n$.*

Proof. From Equation (3.3) we get

$$\text{CPU}(\boldsymbol{R} \triangleright_T \boldsymbol{S}) = \text{CPU}\big(\bigcup_{i=1}^{c/m}(\{\boldsymbol{R}_{j*m-m+1}, \ldots, \boldsymbol{R}_{j*m}\} \triangleright_T^{\mathcal{DIP}} \boldsymbol{S})\big).$$

Remember that during a *$\mathcal{DIP}$Merge* for $\triangleright_T^{\mathcal{DIP}}$ no backtracking is needed. Since $\boldsymbol{S}$ is not partitioned, the number of iterations is at most $m * \frac{n}{c} + (m-1) * n + n$, i.e., the cost for scanning $\{\boldsymbol{R}_{j*m-m+1}, \ldots, \boldsymbol{R}_{j*m}\}$, plus $m-1$ partition switches for each inner tuple, plus the cost for scanning $\boldsymbol{S}$. Thus, the number of tuple comparisons in the worst case is:

$$\begin{aligned}\text{CPU}(\boldsymbol{R} \triangleright_T \boldsymbol{S}) &= \sum_{1}^{c/m}(m * \tfrac{n}{c} + (m-1) * n + n) \\ &= n + c * n \\ &\approx c * n\end{aligned}$$

In terms of unproductive comparisons we have 0 result tuples in the worst case and get: $\text{CPU}(\boldsymbol{R} \triangleright_T \boldsymbol{S}) - |\boldsymbol{R} \triangleright_T \boldsymbol{S}| = c * n$ unproductive comparisons. □

Lemma 12 asserts that for computing a temporal antijoin $\mathcal{DIP}$ limits the number of comparisons per tuple to the number of partitions. State of the art techniques [DBG12], instead, make a nested-loop for computing the intervals of the output tuples. This yields a quadratic number of comparisons.

I/O Cost

This section quantifies the number of block I/Os for computing a temporal anti-join using $\mathcal{DIP}$. Again, we assume that the $\mathcal{DIP}$ partitions are equally sized, i.e., $b = \frac{B}{c}$.

Lemma 13. *Consider relation $\boldsymbol{R}$ partitioned into c $\mathcal{DIP}$ partitions, and relation $\boldsymbol{S}$. The number of I/Os for computing $\boldsymbol{R} \triangleright_T \boldsymbol{S}$ using $\mathcal{DIP}$ is $\frac{c}{m} \times B$.*

Proof. From Equation 3.3, we get

$$\text{IO}(\boldsymbol{R} \triangleright_T \boldsymbol{S}) = \text{IO}\big(\bigcup_{i=1}^{c/m} (\{\boldsymbol{R}_{i*m-m+1}, \ldots, \boldsymbol{R}_{i*m}\} \triangleright_T^{\mathcal{DIP}} \boldsymbol{S})\big)$$

With equally sized partitions, we get:

$$\begin{aligned} &= \tfrac{c}{m} \times \underline{\text{IO}}(\{\boldsymbol{R}_{i*m-m+1}, \ldots, \boldsymbol{R}_{i*m}\} \triangleright_T^{\mathcal{DIP}} \boldsymbol{S}) \\ &= \tfrac{c}{m} \times \big(b * m + B\big) \\ &= B + \tfrac{c}{m} \times B \\ &\approx \tfrac{c}{m} \times B \end{aligned} \tag{3.4}$$

□

When computing $\boldsymbol{R} \triangleright_T \boldsymbol{S}$ with $\mathcal{DIP}$, independent of the number of partitions, $\boldsymbol{R}$ is scanned only once, while $\boldsymbol{S}$ is scanned $\frac{c}{m}$ times. Overall, the cost of our approach is linear with the number of partitions c. In addition, processing m outer partitions simultaneously further reduces the number of I/Os by a factor of m.

3.6.3 Temporal Aggregation

A temporal aggregation $\vartheta_F^T(\boldsymbol{R})$ returns, for each maximal interval during which a set of $\boldsymbol{R}$ tuples is valid, the result of an aggregation function F. For example, in Figure 3.16 the average price of the luxury suites booked in hotel $\boldsymbol{R}$ is computed. The first output row says that, between time 1 and 5, the average price is 80 \$. Note that, due to change preservation [DBG12], two different

$\vartheta^T_{avg(\$)}(\boldsymbol{R})$

T	$avg(\$)$
$[1,5)$	80
$[6,7)$	60
$[7,8)$	71.6
$[8,10)$	75
$[10,11)$	75

Figure 3.16: Temporal aggregation *avg* applied to relation $\boldsymbol{R}$

tuples are returned for $[8,10)$, and $[10,11)$ because, even if their aggregation value is the same, their lineage is different.

A temporal aggregation $\vartheta^T_F(\boldsymbol{R})$ on a table can be decomposed into a sequence of Full Outer Joins between its $\mathcal{DIP}$ partitions:

$$\vartheta^T_F(\boldsymbol{R}) \iff \pi_{T,F'}(\boldsymbol{R}_1 \overset{\mathcal{DIP}}{\underset{T}{\bowtie}} \boldsymbol{R}_2 \overset{\mathcal{DIP}}{\underset{T}{\bowtie}} \cdots \overset{\mathcal{DIP}}{\underset{T}{\bowtie}} \boldsymbol{R}_c) \tag{3.5}$$

The proof of this equivalence is given in Appendix A.

As shown in Figure 3.17, the first partition is full outer joined with the second partition. Afterwards, the intermediate result is full outer joined with the third partition, etc. In other words, $c-1$ $\mathcal{DIP}$*Merge*s are computed. Finally, for each result tuple, the aggregation function F' aggregates the c values using the same aggregation as F (e.g., AVG).

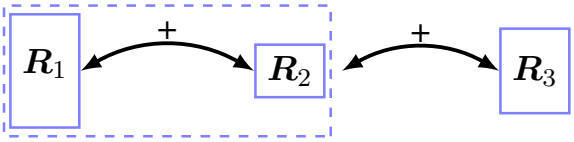


Figure 3.17: A temporal aggregation is computed by (full-outer) joining at linear cost the $\mathcal{DIP}$ partitions.

Example 12. We use Equation (3.5) to transform $\vartheta^T_{avg(\$)}(\boldsymbol{R})$ of our running example to $\Pi_{T,\text{AVG}(\boldsymbol{R}_1.\$,\boldsymbol{R}_2.\$,\boldsymbol{R}_3.\$)}(\boldsymbol{R}_1 \overset{\mathcal{DIP}}{\underset{T}{\bowtie}} \boldsymbol{R}_2 \overset{\mathcal{DIP}}{\underset{T}{\bowtie}} \boldsymbol{R}_3)$. Without loss of generality, we consider only the attributes needed to compute the aggregation, i.e., T and \$. The first full outer join yields $\boldsymbol{R}_1 \overset{\mathcal{DIP}}{\underset{T}{\bowtie}} \boldsymbol{R}_2 = \{([1,5),80,null),([6,7),60,null),([7,8),60,80),([10,11),80,null)\}$. Those tuples are further joined to $\boldsymbol{R}_3$ producing the result shown in Figure 3.18. The projection $\Pi_{T,\text{AVG}(\boldsymbol{R}_1.\$,\boldsymbol{R}_2.\$,\boldsymbol{R}_3.\$)}$ outputs, for each time interval in the result, the average of the three prices, which corresponds to the result in Figure 3.16.

$\boldsymbol{R}_1 \mathbb{X}\!\!\lhd\!\!\rhd_T^{\mathcal{DIP}} \boldsymbol{R}_2 \mathbb{X}\!\!\lhd\!\!\rhd_T^{\mathcal{DIP}} \boldsymbol{R}_3$

T	$\boldsymbol{R}_1.\$$	$\boldsymbol{R}_2.\$$	$\boldsymbol{R}_2.\$$
$[1,5)$	80	null	null
$[6,7)$	60	null	null
$[7,8)$	60	80	75
$[8,10)$	null	null	75
$[10,11)$	80	null	70

Figure 3.18: Full outer join between the $\mathcal{DIP}$ partitions of $\boldsymbol{R}$.

A temporal full outer join between $\boldsymbol{R}_i$ and $\boldsymbol{R}_{i+1}$ returns all join matches ($\boldsymbol{R}_i \bowtie_T \boldsymbol{R}_{i+1}$) plus all anti-join matches of $\boldsymbol{R}_i \rhd_T \boldsymbol{R}_{i+1}$ and of $\boldsymbol{R}_{i+1} \rhd_T \boldsymbol{R}_i$. Each full outer join of the sequence, and not just the first, can be computed without backtracking. This is so because the result of a full outer join between two $\mathcal{DIP}$ partitions is also a $\mathcal{DIP}$ partition: it does not generate tuples with overlapping timestamps.

CPU Cost

Lemma 14. *The number of unproductive comparisons for a temporal aggregation on relation $\boldsymbol{R}$ is upper bounded by $c \times n$.*

Proof. Consider Equation (3.5). Since the projection π can be computed on the fly while writing the result tuples (without doing additional comparisons) we get:

$$\text{CPU}(\vartheta_F^T(\boldsymbol{R})) = \text{CPU}(\boldsymbol{R}_1 \mathbb{X}\!\!\lhd\!\!\rhd_T^{\mathcal{DIP}} \boldsymbol{R}_2 \mathbb{X}\!\!\lhd\!\!\rhd_T^{\mathcal{DIP}} \ldots \mathbb{X}\!\!\lhd\!\!\rhd_T^{\mathcal{DIP}} \boldsymbol{R}_c)$$

When computing a temporal aggregation using full outer joins, a comparison between r and s is unproductive if the tuples do not overlap, since such a comparison only adds NULL values to the result (which do not change the aggregate result). Remember that $c-1$ full outer joins are computed. Since the highest cardinality of a temporal aggregation is $2n-1$ [KS95] (i.e., the number of different T_s and T_e values - 1), in the worst case most of those $2n-1$ intervals are produced by the first full outer join, and the remaining $c-2$ joins perform about $2n-1$ unproductive comparisons each. Thus, we get $(c-2) \times (2n-1) \approx c \times n$ unproductive comparisons. □

Figure 3.19 illustrates such a worst case scenario for computing a temporal aggregation using $\mathcal{DIP}$, with $n = 8$ tuples and $c = 3$ partitions. The first full outer join produces $13 \approx 2n - 1$

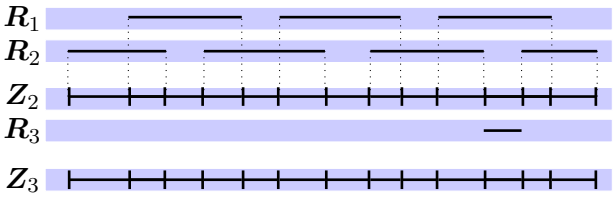


Figure 3.19: Highest cardinality for a full outer join $\boldsymbol{Z}_2 = \boldsymbol{R}_1 \tilde{\bowtie}_T^{\mathcal{DIP}} \boldsymbol{R}_2$. For the full outer join $\boldsymbol{Z}_3 = \boldsymbol{Z}_2 \tilde{\bowtie}_T^{\mathcal{DIP}} \boldsymbol{R}_3$ all comparisons except one are unproductive since they do not change the aggregate value.

intervals, and the second does $13 \approx 2n - 1$ unproductive comparisons (including the one with the last lead) since only one overlapping tuple exists in $\boldsymbol{R}_3$.

I/O Cost

Lemma 15. *The number of I/Os for computing a temporal aggregation $\vartheta_F^T(\boldsymbol{R})$ using $\mathcal{DIP}$ is upper-bounded by $c \times B$.*

Proof. Consider Equation (3.5). Since the projection π can be computed on the fly while writing the result tuples (without additional I/Os) we get:

$$\text{IO}(\vartheta_F^T(\boldsymbol{R})) = \\ \text{IO}(\boldsymbol{R}_1 \tilde{\bowtie}_T^{\mathcal{DIP}} \boldsymbol{R}_2 \tilde{\bowtie}_T^{\mathcal{DIP}} \ldots \tilde{\bowtie}_T^{\mathcal{DIP}} \boldsymbol{R}_c).$$

For the first full outer join, b blocks are read for the outer input, and b blocks for the inner one. In the worst case, $2(|\boldsymbol{R}_1| + |\boldsymbol{R}_2|) = 2(\frac{n}{c} + \frac{n}{c}) = 4\frac{n}{c}$ tuples are returned and $4 \times b$ blocks are needed for storing this intermediate result. For the second join, $4 \times b$ blocks are read for the outer input and b for the inner. In the worst case, $6 \times b$ blocks are needed for storing the intermediate result. Generalizing, for the $(c-1)$-th full outer join, i.e., the last one to compute, we read in the worst case $2 \times (c-1) \times b$ blocks for the outer input and b for the inner, and we write $2 \times c \times b$ blocks

for the result. Summing up the I/Os of all $c-1$ full outer joins we get:

$$\begin{aligned}\text{IO}(\vartheta_F^T(\boldsymbol{R})) =& \sum_1^{c-1}(2 \times i \times b + b + 2 \times (i+1) \times b) \\ =& 2 \times b \sum_1^{c-1} i + \sum_1^{c-1} b + 2 \times b \sum_1^{c-1}(i+1)\end{aligned}$$

Since $\sum_1^{c-1} i = \frac{(c-1)c}{2}$ and $\sum_1^{c-1}(i+1) = \frac{c(c+1)-2}{2}$, we obtain

$$\begin{aligned}&2\frac{(c-1)c}{2} \times b + (c-1)b + 2\frac{c(c+1)-2}{2} \times b = \\ &2 \times c^2 b + (c-3)b = 2c \times B + B - 3b \simeq c \times B\end{aligned}$$

□

The I/O cost for computing a temporal aggregation is linear in the number of the partitions. Furthermore, opposite to the state of the art approaches, such as the Aggregation Tree [KS95], the Balanced Tree [MFVLI03], and the SB-Tree [YW03], our approach is not limited to distributive aggregates: standard deviation, for example, is also computable using $\mathcal{DIP}$.

3.7 Implementation

In this section we discuss our implementation. First we describe how to implement each temporal operator in the executor of the DBMS using a sequence of $\mathcal{DIP}$*Merge*s. Then we propose an efficient implementation of $\mathcal{DIP}$*Merge* itself, i.e., an algorithm computing joins, anti-joins, and full outer joins without backtracking.

3.7.1 Implementing the Temporal Operators

Equations (3.1), (3.3), and (3.5) directly lead to the algorithms in Figure 3.20. In the executor of the DBMS, each temporal $Operator$ is computed by first creating the partitions (i.e., calling $Create\mathcal{DIP}$), and then calling iteratively $\mathcal{DIP}$*Merge* as described below.

Temporal Join $\boldsymbol{R} \bowtie_T \boldsymbol{S}$

```
1  R_1, ..., R_c ← CreateDIP(R)
2  S_1, ..., S_c ← CreateDIP(S)
3  Z = ∅
4  for i = 1 to c do
5  |  k = min(i + m − 1, c)
6  |  for j = 1 to c do
7  |  |  T = DIPMerge({R_i, ..., R_k}, S_j, ⋈)
8  |  |  Z = Z ∪ T
9  |  i = k
10 return Z
```

Temporal Anti-join $\boldsymbol{R} \triangleright_T \boldsymbol{S}$

```
1  R_1, ..., R_c ← CreateDIP(R)
2  Z = ∅
3  for i = 1 to c do
4  |  k = min(i + m − 1, c)
5  |  T = DIPMerge({R_i, ..., R_k}, S, ▷)
6  |  Z = Z ∪ T
7  |  i = k
8  return Z
```

Temporal Aggregation $\vartheta^T_F(\boldsymbol{R})$

```
1  R_1, ..., R_c ← CreateDIP(R)
2  Z ← R_1
3  for i = 2 to c do
4  |  Z = DIPMerge({Z}, R_i, ⟗)
5  Z = Π_{T,F'}(Z)
6  return Z
```

Figure 3.20: Each temporal operator is computed calling multiple times $\mathcal{DIP}Merge$.

For $\boldsymbol{R} \bowtie_T \boldsymbol{S}$, first $\boldsymbol{R}$ and $\boldsymbol{S}$ are partitioned by $CreateDIP$. Then m outer $\mathcal{DIP}$ partitions are $\mathcal{DIP}Merge$d with each inner partition, and the result tuples are collected in $\boldsymbol{Z}$.

For $\boldsymbol{R} \triangleright_T \boldsymbol{S}$, only $\boldsymbol{R}$ is partitioned. Then m outer partitions are $\mathcal{DIP}Merge$d with the entire $\boldsymbol{S}$ relation, and the result tuples are collected in $\boldsymbol{Z}$.

For $\vartheta^T_{f(A)}(\boldsymbol{R})$, the first $\mathcal{DIP}$ partition is $\mathcal{DIP}Merge$d with the second, and the result tuples are collected in $\boldsymbol{Z}$; $\boldsymbol{Z}$ is iteratively $\mathcal{DIP}Merge$d with the subsequent $\mathcal{DIP}$ partitions[1]. Finally, a projection on $\boldsymbol{Z}$ computes the aggregation function F' on the values $A_1, \ldots, A_c$.

3.7.2 Implementation of $\mathcal{DIP}Merge$

Algorithm 2 shows the implementation of $\mathcal{DIP}Merge$. The first argument is a set of $\mathcal{DIP}$ partitions $\{\boldsymbol{R}_1, \ldots, \boldsymbol{R}_m\}$ each with schema $(T, R_1, \ldots, R_k)$. The second argument $\boldsymbol{S}$ is an inner $\mathcal{DIP}$ partitions (or the entire relation) with schema $(T, I_1, \ldots, I_l)$. Finally, Op is the operator to compute, i.e., a temporal join, anti-join, or full outer join. The algorithm computes Op with a single scan of $\{\boldsymbol{R}_1, \ldots, \boldsymbol{R}_m\}$ and $\boldsymbol{S}$, without backtracking.

[1]Our implementation applies standard DBMS optimization techniques, such as projecting on the attributes required for the aggregation (i.e., T and A), so that the full outer join result includes only the needed attributes.

Algorithm 2: $\mathcal{DIPMerge}(\{\boldsymbol{R}_1, \ldots, \boldsymbol{R}_m\}, \boldsymbol{S}, Op)$

Input : $\boldsymbol{R}_i(T, R_1, ..., R_k), \boldsymbol{S}(T, I_1, ..., I_l), Op \in \{\bowtie, \triangleleft, \rtimes\}$
Output: $\boldsymbol{Z}(T, R_1, \ldots, R_k, I_1, \ldots, I_l)$

1 **for** $i = 1$ **to** m **do**
2 $\quad r[i] \leftarrow fetchRow(\boldsymbol{R}_i)$
3 $\quad r[i].X = [-\infty, r[i].T_s)$ `// lead of r[i]`
4 $i = 1$
5 $s \leftarrow fetchRow(\boldsymbol{S})$
6 $s.X = [-\infty, s.T_s)$ `// lead of s`
7 **while** $!null(r[i].T) \vee !null(s.T)$ **do**
8 $\quad$ **if** $Operator = \rtimes$ **then**
9 $\quad\quad$ **if** $len(r[i].X) > 0 \wedge overlap(r[i].X, s.T)$ **then**
10 $\quad\quad\quad \boldsymbol{Z} = \boldsymbol{Z} \cup < (r[i].X \cap s.T), null^k, s.I_1, \ldots, s.I_l >$
11 $\quad$ **if** $Operator \in \{\triangleright, \rtimes\}$ **then**
12 $\quad\quad$ **if** $len(s.X) > 0 \wedge overlap(r[i].T, s.X)$ **then**
13 $\quad\quad\quad \boldsymbol{Z} = \boldsymbol{Z} \cup < (r[i].T \cap s.X), r.R_1, \ldots, r.R_k, null^l >$
14 $\quad$ **if** $Operator \in \{\bowtie, \rtimes\}$ **then**
15 $\quad\quad$ **if** *overlap*$(r[i].T, s.T)$ **then**
16 $\quad\quad\quad \boldsymbol{Z} = \boldsymbol{Z} \cup < (r[i].T \cap s.T), r[i].R_1, ..., r[i].R_k, s.I_1, ..., s.I_l >$
17 $\quad$ **if** $!null(r[i].T) \wedge (null(s.T) \vee r[i].T_e \leq s.T_e)$ **then**
18 $\quad\quad$ **if** $r[i].T_e > r[i].X_s$ **then** $longestR[i] = r[i].T_e$
19 $\quad\quad r[i] \leftarrow FetchRow(\boldsymbol{R}_i)$
20 $\quad\quad$ **if** $!null(r[i])$ **then**
21 $\quad\quad\quad r[i].X = [longestR[i], r[i].T_s)$
22 $\quad\quad$ **else**
23 $\quad\quad\quad r[i].T = null$
24 $\quad\quad\quad r[i].X = [longestR[i], \infty)$
25 $\quad$ **else**
26 $\quad\quad$ **if** $i < m$ **then**
27 $\quad\quad\quad i = i + 1$
28 $\quad\quad$ **else**
29 $\quad\quad\quad i = 1$
30 $\quad\quad\quad$ **if** $s.T_e > s.X_s$ **then** $longestS = s.T_e$
31 $\quad\quad\quad s \leftarrow FetchRow(\boldsymbol{S})$
32 $\quad\quad\quad$ **if** *!null(s)* **then**
33 $\quad\quad\quad\quad s.X = [longestS, s.T_s)$
34 $\quad\quad\quad$ **else**
35 $\quad\quad\quad\quad s.T = null$
36 $\quad\quad\quad\quad s.X = [longestS, \infty)$
37 **return** $\boldsymbol{Z}$

At the beginning, the lead of the current tuple $r[i].X$ in the i-th outer partition is initialized as the interval between $-\infty$ and the starting point of the first tuple. We do the same for the current tuple s in $\boldsymbol{S}$. Initially $i = 1$. During each iteration, the algorithm fetches a new tuple from $\boldsymbol{R}_i$ (line 19). When all tuples in $\boldsymbol{R}_i$ that overlap s have been found, the algorithm switches to partition $\boldsymbol{R}_{i+1}$ (line 27). Once all outer partitions have been checked against s, the algorithm fetches a new $\boldsymbol{S}$ tuple (line 31) and restarts processing $\boldsymbol{R}_1$ from its last scanned tuple. The result tuples change depending on the Op to be computed (lines 8-16):

- **Join:** For the join matches, we directly use Lemma 6 since $r[i]$ and s only join iff they overlap: if tuple $r[i]$ is the last join match of s, then no tuple before $r[i]$ can match with the successor of s. Line 16 outputs the join matches by concatenating the attributes of $r[i]$ and s.

- **Anti-Join:** For the anti-join matches, the lead $s.X$ must be considered. Lemma 6 holds between $r[i].T$ and $s.X$. Line 13 outputs the anti-join matches. Since no $\boldsymbol{S}$ tuple exists for an anti-join result tuple, i.e., during $(r[i].T \cap s.X)$, for each attribute $I_1, \ldots, I_l$ of the inner input a NULL value is returned.

- **Full Outer Join:** To make sure that the full outer join returns a $\mathcal{DIP}$ partition with sorted elements (so that the next full outer join of the sequence does not require any additional sorting), the anti-join matches *must* be written before the join matches. Since the lead $s.X$ (or $r[i].X$) is the interval between s (or $r[i]$) and its predecessor, $s.X$ comes always before $s.T$ (as well as $r[i].X$ comes before $r[i].T$), and an interval overlapping with $s.X$ is written before an interval overlapping with $s.T$.

The algorithm ends when all input tuples have been processed (i.e., when $r[i].T$ and $s.T$ are both null). Note that in case only one input (e.g., $\boldsymbol{S}$) has been scanned entirely, the algorithm goes on to return the anti-join matches of all remaining outer tuples.

3.8 Experiments

For the experiments on disk, we used an Intel(R) Core2 Duo Processor E8600 @ 3.33 GHz machine with 4GB main memory and a 480 GB Hard Disk, running Ubuntu 14.04.3. (L1 cache: 32 KB, L2 cache: 6 MB). For the experiments in main memory, we used a 2 x Intel(R) Xeon(R)

CPU E5-2440 (6 cores each) @ 2.40GHz with 64GB main memory, and running CentOS 6.4 (L1 cache: 192 KB, L2 cache: 1536 KB, L3 cache: 15 MB). For the main memory experiments, all indices and all data are kept in memory and no disk I/O for reading or sorting is done.

We compute the performances of Temporal Alignment (Align, [DBG12]), the TimeLine Index (TimeLine, [KMV+13]), Overlap Interval Partitioning (OIP, [DBG14]), Sort-Merge (SM, [GS91]), the Sweepline algorithm (Sweepline, [APR+98]), the Aggregation Tree (AggTree, [KS95]) Sort-Aggregate (SortAgg, [Gra93]) and $\mathcal{DIP}$. Real world data as well as synthetic data is used. We use the Swiss Feed Data Warehouse [TBBC12] and the Time Interval (TI) dataset [TId15] as real world datasets.

3.8.1 Real World Data

In this subsection we compare the runtimes of the approaches for computing temporal joins, anti-joins, and aggregations. We use the Swiss Feed Data Warehouse and fix the ratio between the length of the history and the number of tuples to 1:1, e.g., a history of 100k granules stores 100k tuples, and we then increase the history length. Intervals have length varying from 1 to 10k granules: 90% of the intervals have length smaller than 10 granules (they represent lab measurements that change over time, and must be repeated frequently); 9.5% of the remaining intervals have length up to 1000 granules; 0.5% up to 10000 granules (they represents lab measurements of values that remain constant, and are repeated seldom). We vary those parameters in the experiments in Subsection 3.8.2.

Temporal Joins

First, we compute a temporal join that joins the values of two different nutritive values (Protein and Fat). The runtime is measured for disk-based computations and for in-memory computations.

Execution on disk. Figure 3.21(a) shows that Align performs badly when the data history grows, since it checks $|\boldsymbol{R}| \times |\boldsymbol{S}|$ comparisons. The TimeLine index performs better since it avoids unproductive comparisons, however each result tuple (r, s) is produced by making one index look-up in $\boldsymbol{R}$ and one in $\boldsymbol{S}$. Copying the entire dataset into main memory is not beneficial for large input relations since each index look-up fetches a different memory block. Finally, long lived tuples, (e.g., 10k granules long) are fetched multiple times with one index look-up for each

tuple they match. Sweepline does not perform well on disk since the active tuples have to be updated when the sweepline advances. This is expensive for disk-resident data.

In Figure 3.21(b) we show approaches that scale better on disk, and can handle more data. OIP performs worse than $\mathcal{DIP}$ and SM because of the many short intervals present in the dataset. Those tuples are a bottleneck for OIP since they make the nested-loop between the partitions very expensive in terms of unproductive comparisons: with 5M tuples, 6.5×10^8 combinations are checked by OIP, 4×10^7 by SM, and only 1×10^5 by $\mathcal{DIP}$.

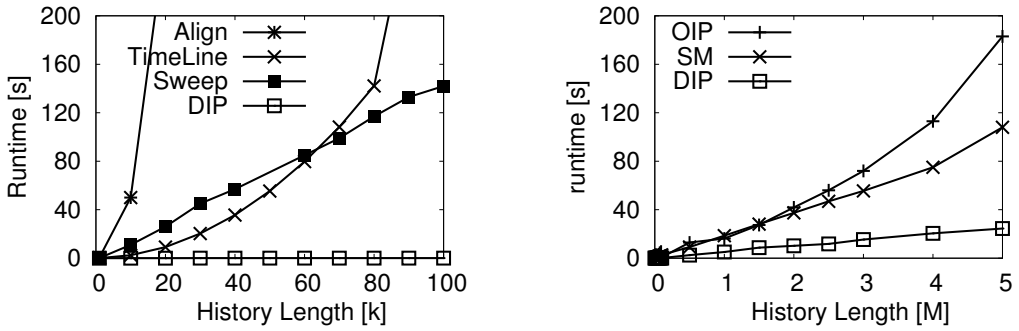


Figure 3.21: Temporal join on on disk

Execution in main memory. Figure 3.22 shows that all approaches benefit from an in-memory execution as expected, except from the TimeLine index, which always keeps the data in memory. Figure 3.22(b) shows that the runtime of OIP, SM, and $\mathcal{DIP}$ is proportional to the amount of unproductive comparisons: with 25M tuples, 2×10^{10} unproductive comparisons are done by OIP, 1.2×10^9 by SM, and 2.4×10^6 by $\mathcal{DIP}$. In Figure 3.22(c), we show that for a history of 300M tuples, $\mathcal{DIP}$ is more than two minutes faster than Sweepline. This is so because, although Sweepline does at most one unproductive comparison per tuple, the list of active tuples is allocated and deallocated at run time yielding a poor memory locality. Computing a random memory access per active tuple makes the join computation expensive for Sweepline. Figure 3.22(d) shows that, if the sorting (for Sweepline) and the partitioning (for $\mathcal{DIP}$) are computed offline, $\mathcal{DIP}$ computes the join one order of magnitude faster than Sweepline. Our results confirm the experimental evaluation by Stroustrup in [Str12].

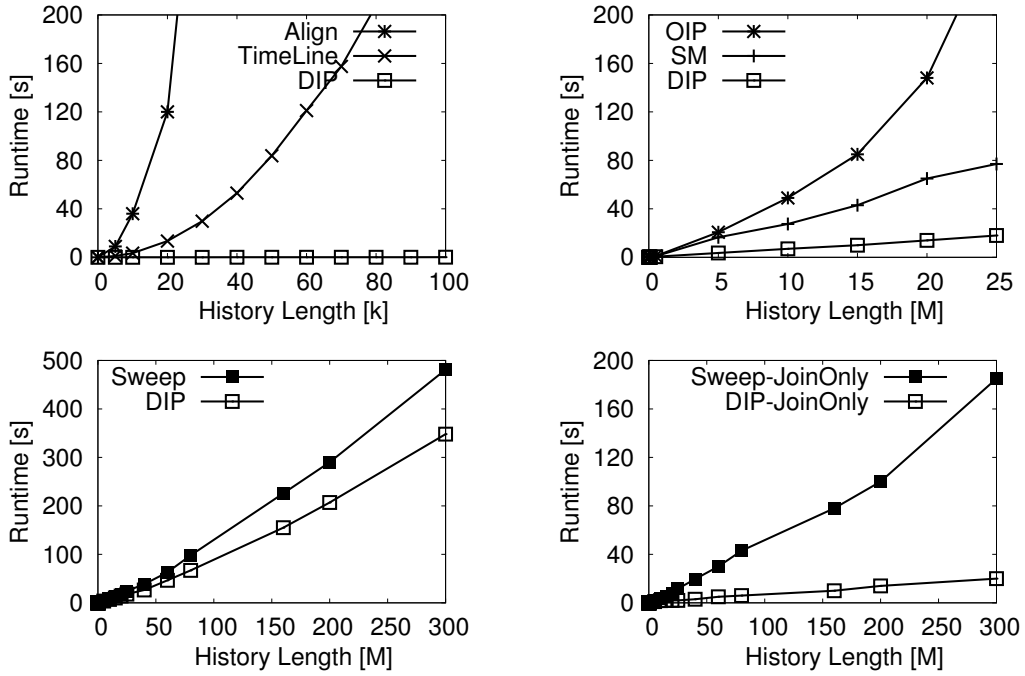
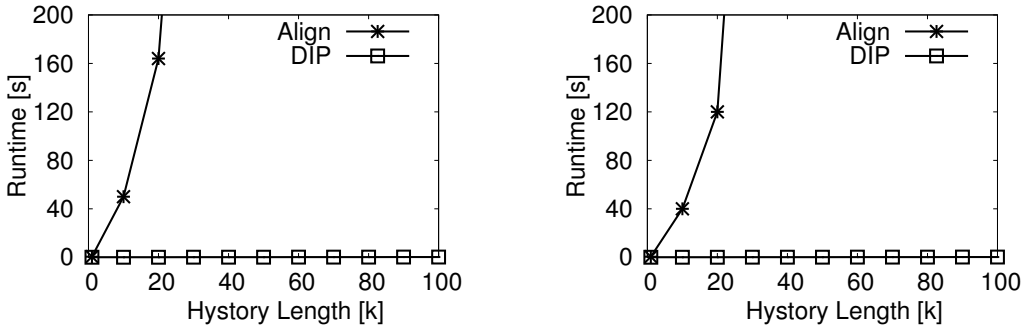


Figure 3.22: Temporal join in main memory

Temporal Anti-Joins

In this experiment we compute a temporal anti-join to find all intervals for which a protein measurement but no fat measurement exists. To the best of our knowledge, only Dignös et al. [DBG12] provide a solution for computing temporal anti-joins. The nested-loop with which the alignment operator is computed is however expensive, since query optimizers are not able to use interval T to optimize the query plan. Figure 3.23 shows that the runtime of alignment on disk is similar to the runtime in main memory because a small dataset, once it has been fetched from disk, is cached in main memory. However, checking n^2 combinations is expensive even in main memory. $\mathcal{DIP}$ provides the first non-quadratic solution for computing temporal anti-joins.

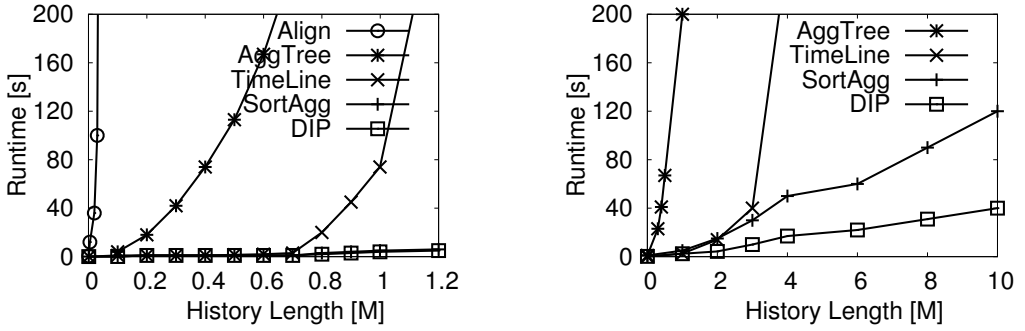


(a) Disk (b) Memory

Figure 3.23: Temporal anti-join.

Temporal Aggregation

This experiment reports the runtime for the computation of a temporal aggregation, i.e., we compute the average value for the measurements stored in the Swiss Feed Data Warehouse. The Aggregation Tree is not efficient and does not scale even with high memory availability (Figure 3.24.b). The TimeLine Index performs like $\mathcal{DIP}$ as long as the entire dataset can be kept



(a) Disk (b) Memory

Figure 3.24: Temporal aggregation.

in memory. If the data does not fit into memory, the index look-ups require additional I/Os from disk with a higher cost. This makes the approach inefficient: for more than 3M tuples, TimeLine requires[2] more than 64 GB of main memory and becomes inefficient on our machines. $\mathcal{DIP}$ does not require an index and stays stable. For the experiments in main memory, the runtime of our approach grows linearly with the length of the data history (Figure 3.24.b). Sort-aggregate

[2]The authors' implementation has been used for the experiments.

requires backtracking and performs slower than $\mathcal{DIP}$. For 4M to 6M tuples, sort-aggregate stays almost constant because few long lived tuples occur between the 4-th and the 6-th granule in the dataset; afterwards it increases its runtime because of longer backtracking.

TI Dataset

The TI dataset [TId15] is public, and stores the Universal Resource Identifiers (URIs) for the time intervals commonly used by the UK Government. Tuples are stored as $< T_s, T_s, \text{URI} >$ pairs. The time granularity is expressed in number of days. The intervals have length 1 (i.e., one day), {28,29,30,31} (i.e., one month), {365,366} (i.e., one year), {547,548} (i.e., one and a half year),{730,731} (i.e., two years). Figure 3.25 shows the runtime for computing a self-join on disk and in main-memory using the TI dataset. In memory, $\mathcal{DIP}$ is six times faster than Sweepline, and over an order of magnitude fatser than the other approaches. On disk, Sweepline deteriorates since for each tuple the file storing the active tuples must be rewritten entirely (the URIs have different length). $\mathcal{DIP}$ is the only approach that is robust both if the dataset is memory- and if it is disk-resident. It accesses the tuples sequentially and, at the same time, keeps the number of unproductive comparisons low.

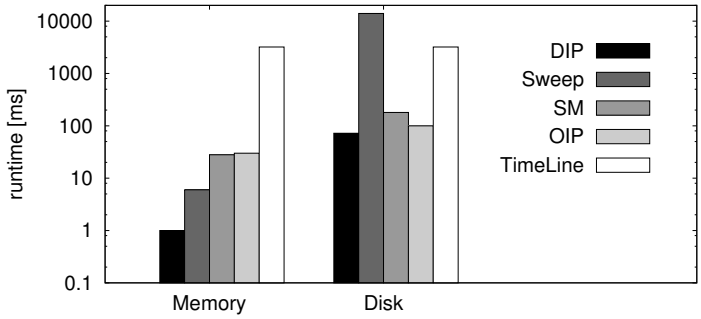


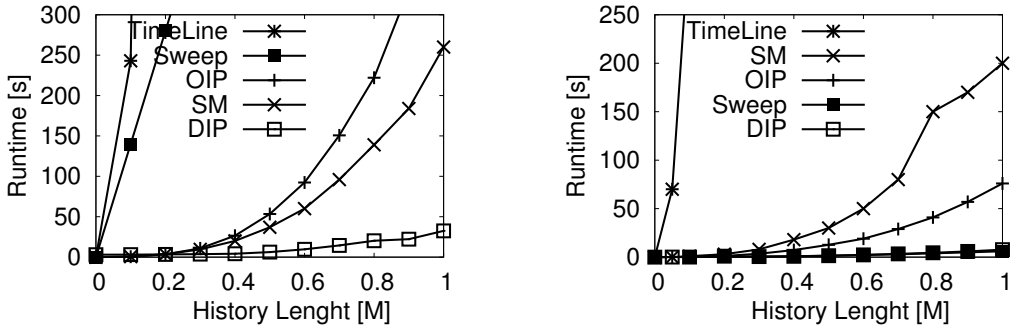
Figure 3.25: Temporal Join for the TI dataset.

3.8.2 Synthetic data

In this subsection we use synthetic data, and evaluate the approaches by varying the characteristics of the data history. We first increase the number of partitions by increasing the number of tuples valid as time passes by. Then, we show the effect of processing m partitions simultaneously for the average and worst case scenario.

Size of Dataset

This experiment shows how the approaches behave when the number of tuples valid as the time passes by increases, i.e., when recently more data are collected compared to the past. For each 100k time granules in the history, 100k more tuples exist compared to the previous 100k time granules (e.g., from the 0-th to 100k-th time granule of the history we have 100k tuples; from the 100k-th to 200k-th granule we have 200k tuples; from the 200k-th to 300k-th granule we have 300k tuples; etc.).



(a) Disk (b) Memory

Figure 3.26: Increase of the number of tuples collected throughout of the data history

In Figure 3.26(a), we can see that $\mathcal{DIP}$ scales best. This is so because $\mathcal{DIP}$ is not affected by the size of the partitions: for $c_{\boldsymbol{R}}$ and $c_{\boldsymbol{S}}$ $\mathcal{DIP}$ partitions, the amount of unproductive comparisons of $\mathcal{DIP}$ does not change if the partitions are equally sized or if they are unbalanced. For OIP, if the partitions are unbalanced, the unproductive comparisons increase. Sweepline performs as well as $\mathcal{DIP}$ for an in-memory execution since the history length is limited to 1M granules. For larger datasets (cf. Figure 3.22), i.e., after many insertions and deletions, the list of active tuples becomes scattered in memory and the approach slows down. $\mathcal{DIP}$ is also robust if the partitions are stored on disk.

Varying m in the Average Case for $\mathcal{DIP}$

In this experiment, we show how $\mathcal{DIP}$ behaves in the general case for different values of m. Partitions are large (i.e., 1M tuples each) and they do not fit into the cache. In Figure 3.27, we show that the runtime of a join increases only by an order of two when m grows since, as shown in Equation (3.2a), relation $\boldsymbol{R}$, independent of the value of m, must be scanned c times.

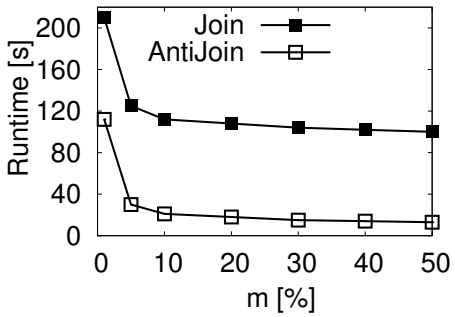


Figure 3.27: Increase of m for a join and an anti-join.

The number of scans of $\boldsymbol{S}$, instead, is reduced by a factor of m. For an anti-join, instead, $\boldsymbol{R}$ is scanned only once, therefore when the number m of outer partitions processed simultaneously increases, the number of times $\boldsymbol{S}$ is scanned decreases (Equation 3.4). Figure 3.27 shows an improvement of the performances of an order of magnitude.

Varying m in the Worst Case for $\mathcal{DIP}$

In this experiment, we show the worst case for computing a join using $\mathcal{DIP}$. This happens if all tuples overlap, and each tuple is placed in a new partition. Note that this means there is a time point where all data is valid, which is not usually the case for temporal databases. In this experiment, since each $\boldsymbol{R}$ tuple overlaps with all $\boldsymbol{S}$ tuples, all approaches are quadratic.

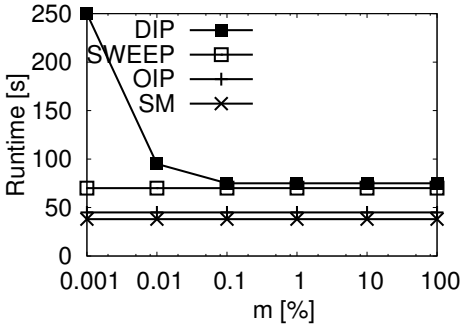


Figure 3.28: High number of $\mathcal{DIP}$ partitions.

In Figure 3.28, we show that our approach requires little memory to become competitive in a worst case scenario. The graph shows that as soon as 0.1 % of the outer partitions are processed simultaneously, $\mathcal{DIP}$ reaches the performance of the Sweepline approach. This is so for two

reasons (cf. Equation 3.2): *i)* small outer partitions are entirely cached and can be reused for the next $\mathcal{DIP}$*Merge*; *ii)* when m grows, the number of scan on $\boldsymbol{S}$ decreases by a factor of m. OIP and SM are slightly faster in a worst case scenario since the tuples of a relation (for SM) and of a partition (for OIP) are accessed sequentially, while for $\mathcal{DIP}$ and for Sweepline tuples are accessed randomly since each tuple is in a different partition (for $\mathcal{DIP}$) and each active tuple in a different memory block (for Sweepline).

3.9 Conclusions and Future Work

In this Chapter we have proposed *Disjoint Interval Partitioning* ($\mathcal{DIP}$). $\mathcal{DIP}$ partitions a temporal relation into the minimum number c of partitions storing non-overlapping tuples. $\mathcal{DIP}$ is a new and general approach that makes sort-based operator efficient in the presence of interval data. We have proved that temporal joins, anti-joins, and aggregation are computed with at most c unproductive comparisons per tuple, independently of the size of the partitions. We have empirically shown that $\mathcal{DIP}$ outperforms the state of the art solutions when computing temporal operators over historical data.

As a future work, we are interested to: *i)* incrementally update the $\mathcal{DIP}$ partitions: if a new tuple r is stored in the database and its timestamp is in the past, then checking only the last tuple of the partitions does not ensure that r is disjoint from all other tuples; *ii)* efficiently incorporating conditions over non-temporal attributes: while for a temporal equijoin they can be trivially computed on the fly, for anti-joins it becomes complex to generate the *leads* since their starting point depends on the previously scanned tuple that has the same non-temporal values.

3.10 Acknowledgments

This work has been developed in the context of the Tameus project between the University of Zurich and Agroscope, with funding from the Swiss National Science Foundation. We thank Jerinas Gresch from Siemens for contributing to the implementation of $\mathcal{DIP}$*Merge*.

Appendix

Proof of Equivalence Rule

Now we prove that $\vartheta_F^T(\boldsymbol{R})$ gives the same result as Equation (3).

Lemma 16. *A Temporal Aggregation on an input relation $\boldsymbol{R}$ can be decomposed as the full outer join between its $\mathcal{DIP}$ partitions:*

$$\vartheta_F^T(\boldsymbol{R}) = \Pi_{T,F'}(\boldsymbol{R}_1 \rtimlt_T^{\mathcal{DIP}} \boldsymbol{R}_2 \rtimlt_T^{\mathcal{DIP}} \ldots \rtimlt_T^{\mathcal{DIP}} \boldsymbol{R}_c)$$

where F' is an aggregation function that has the same semantic as F but applies to columns rather than to rows.

Proof. Proof by induction. We rewrite the sequence of full outer joins in the equivalence rule as:

$$\boldsymbol{Z}_n = \begin{cases} \boldsymbol{R}_1, & \text{if } n = 1 \\ \boldsymbol{Z}_{n-1} \rtimlt_T^{\mathcal{DIP}} \boldsymbol{R}_n, & \text{if } 2 \leq n \leq p \end{cases}.$$

We check that each conjunction of the definition of $\vartheta_F^T(\boldsymbol{R})$ in Table 3.2 is satisfied by $\boldsymbol{Z}_n$, with the hypothesis that $\boldsymbol{Z}_{n-1}$ satisfies it:

1. *for each $w \in \boldsymbol{Z}_n$, $w.T_s$ and $w.T_e$ correspond to the starting or ending point of two tuples $r, s \in \boldsymbol{R}$, i.e., $w.t_s = (r.T_s \vee r.T_e) \wedge w.t_e = (s.T_s \vee s.T_e)$*

   $n = 1$ Since $\boldsymbol{R}_1 = \boldsymbol{R}$ then $\forall w \in \boldsymbol{Z}_1 \Rightarrow (\exists r \in \boldsymbol{R} : w.T_s = r.T_s \wedge w.T_e = r.T_e)$,which satisfies condition 1 for $r = s$.

   $n > 1$ Remember that $\boldsymbol{Z}_{n-1} \rtimlt_T \boldsymbol{R}_n$ corresponds to the union of the Join and of the anti-joins between $\boldsymbol{Z}_{n-1}$ and $\boldsymbol{R}_n$, and viceversa. We now show that condition 1 holds for each of those three joins. For $\boldsymbol{Z}_{n-1} \bowtie_T \boldsymbol{R}_n$, given an overlapping pair (z, r), a result interval is $w.T = [max(z.T_s, r.T_s), min(z.T_e, r.T_e))$: $z.T_s$ and $z.T_e$ by hypothesis satisfy condition 1; since $\boldsymbol{R}_n$ is a partition (i.e., a selection) of $\boldsymbol{R}$, then $r.T_s$ and $r.T_e$ also satisfy condition 1 (for $r = s$). For $\boldsymbol{Z}_{n-1} \rhd_T \boldsymbol{R}_n$, a result interval is $w.T = [z.T_s, z.T_e)$ if no overlapping tuple in $\boldsymbol{R}_n$ exists (which by hypothesis hold condition 1); if a tuple $r_j \in \boldsymbol{R}_n$ exists such that $overlap(z, r_j)$, then a result interval can be

i) $w.T = [z.T_s, r_j.T_s)$, *ii)* $w.T = [r_j.T_e, z.T_e)$, *iii)* $w.T = [r_j.T_e, r_{j+1}.T_s)$. All these intervals satisfy condition 1. Analogous for $\boldsymbol{R}_n \triangleright_T \boldsymbol{Z}_{n-1}$.

2. *for each* $w \in \boldsymbol{Z}_n$*, there must not exist in* $\boldsymbol{R}$ *a tuple that starts or ends within* $w.T$*, i.e.,* $\forall u \in \boldsymbol{R}\big(overlap(u, w) \leftrightarrow (w.T - u.T = \emptyset)\big)$.

$n = 1$ By definition a $\mathcal{DIP}$ partition does not store overlapping tuples: given $w \in \bar{\boldsymbol{R}}_1$, a tuple $u \in \boldsymbol{R}_1$ with $w.T_s \leq u.T_s \leq w.T_e$ or $w.T_s \leq u.T_e \leq w.T_e$ cannot exist.

$n > 1$ For $\boldsymbol{Z}_{n-1} \bowtie_T \boldsymbol{R}_n$ and $\boldsymbol{Z}_{n-1} \triangleright_T \boldsymbol{R}_n$ condition 2 holds since the timestamp of each result tuple w is a sub-interval of a tuple $z \in \boldsymbol{Z}_{n-1}$ (which, by hypothesis, satisfies condition 2). For $\boldsymbol{R}_n \triangleright_T \boldsymbol{Z}_{n-1}$, the timestamp of each result tuple w is the sub-interval of $r \in \boldsymbol{R}_n$ during which no tuple in $\boldsymbol{Z}_{n-1}$ exists. This means that in all previous $\mathcal{DIP}$-partitions no tuple existed during $w.T$. Since the union of all the $\mathcal{DIP}$ partitions gives $\boldsymbol{R}$, then no tuple u exists in $\boldsymbol{R}$ overlapping w other than itself.

3. *for each* $w \in \boldsymbol{Z}_n$*, the set* $\boldsymbol{R}$*' of all tuples valid over* $w.T$ *must be returned in the join result, i.e.,* $\forall u \in \boldsymbol{R}\big(overlap(u, w) \leftrightarrow u \in \boldsymbol{R}'\big)$

$n = 1$ By definition $r.T$ stores the interval of validity of r.

$n > 1$ The full outer join returns, by definition, the tuple of $\boldsymbol{R}_n$ overlapping $w.T$. If no tuple overlapping $w.T$ exists in $\boldsymbol{R}_n$, it returns a $null$ value.

□

CHAPTER 4

Feedbase.ch: a Data Warehouse System for Animal Feed

Abstract

In Switzerland we have developed Feedbase.ch, a system that uses data from the Swiss Feed Data Warehouse (SFDW) to monitor and assess the quality of animal feed. Opposite to other Feed Data Warehouses, our system provides to the user accurate temporal (e.g., measurements are stored with up to a daily granularity), spatial (e.g., we store the coordinates of the field where the feed has been grown and sampled) and biological information (e.g., the stage of maturity of the feed) to increase the level of detail during the data analysis, since the nutritive content of the animal feeds is dependent on those dimensions. In this chapter we describe the architecture, interface, and functionalities of Feedbase.ch and the lessons learned for minimizing the response time to users.

Although single dimensions are non-selective (e.g., many measurements refer to the same feed), the data cube of most data warehouses is sparse, i.e., for many combinations of dimensional values no data exists in the fact table. For example, in the SFDW nutrients are not measured for all feeds in all Swiss locations on a daily basis since chemical analysis are expensive. Thus,

for guiding the user through meaningful selections, i.e., selections for which data exist in the fact table, we use data-driven menus. Data-driven menus are used in Feedbase.ch for limiting the dimensional values to display to the user and for which data must be retrieved from the fact table. An option in a menu is shown to the user only if, for that dimensional value, data exists in the fact table satisfying the selections in the previous menus. Systems dealing with feed DWs either provide menus that are not data-driven, or suffer from a high runtime in implementing them because of the join between the fact table and the dimensions (the values to be shown in the menus are stored in the dimensions, while the measures whose presence must be checked are stored in the fact table). Such joins are expensive since the dimensional keys in the fact table are not selective and they end up fetching many tuples. Computing such menus on a subset of the dataset is not a good strategy since it would exclude some options for which data exist, instead. We describe the challenges and solutions for implementing those menus efficiently using *partial evaluation*.

For densifying the sparse data cube of the SFDW, we introduce *derived facts*. In the SFDW, derived facts calculate the value of nutrients that have not been analysed in the lab, and thus are not stored in the fact table. Derived facts are computed online by Feedbase.ch using chemical formulas defined on the value of other nutrients. Since some of those nutrients might also not have been measured on a given feed sample, equijoins cannot be used to retrieve the measurements needed for computing derived facts. Derived facts are computed in SQL as a series of Nearest Neighbour Joins, where the i-th Nearest Neighbour Join retrieves the value of the i-th nutrient appearing in the formula. We compare different query plans for computing derived facts, and experimentally show that the one based on Group- and Selection-enabled Nearest Neighbour Joins is the most efficient since it avoids that blocks of the fact table are accessed multiple times.

Since single dimensions are not selective, computing aggregations over the fact table when only few dimensions are constrained is expensive because many measurements share the same keys. Our system uses *materialized views* for efficiently computing distributive aggregations over the fact table. We compare the performance gain obtained using materialized views in computing detailed statistics in the SFDW, against query plans based on a normalized and on a denormalized fact table.

We finally describe three real use cases of Feedbase.ch.

4.1 Introduction

The Swiss Feed Data Warehouse (SFDW) has been built by the Database Technology group of the University of Zurich in collaboration with *Agroscope*, the Swiss Federal organization for agriculture, food and environmental research. It stores the history for the last 40 years of the nutritive content of the animal feed in Switzerland. Detailed biological, spatial, and temporal information are stored for the feed samples analyzed and collected in the SFDW. Although single dimensions are non-selective (e.g., many measurements exist for feed 'Hay'), the data cube of the SFDW is sparse since for many combinations of dimensional keys no data exists in the fact table (nutrients are not measured on a daily basis for all feeds from every Swiss location). It is therefore necessary to help the user to select the options for which data exist in the SFDW.

Example 13. Consider the creation of a set of data-driven menus, where each menu makes selections on one particular dimension of the star schema. In Feedbase.ch (the web application on top of the SFDW), the user first restricts the *Feed* dimension, second the *Nutrient* dimension, then the *Time*, etc. In a data-driven menu, the options to display strictly depend on the options selected by the user in the previous menus. For example, in Feedbase.ch we want the *Nutrients* menu to display just the nutrients for which measurements exist in the fact table for the *Feeds* selected previously by the user. Traditionally, a join between the *Feed* dimension and the fact table is required [Kes08; WQZ+15] to get the data that exist for the selected options (since the feed names are stored in the dimension table). However, in a DW single dimension keys are not selective: for a given feed, many measurements exist in the fact table, and the join is expensive.

For computing a data-driven menu with interactive response time (i.e., in less than one second), we introduce queries based on *partial evaluation*. For the $(i+1)$-th data-driven menu, we re-use part of the result of the queries that computed the 1-st, ..., $i-$th menus, to not redundantly join the fact table with the 1-st, ..., i-th dimensions. By applying partial evaluation, for the $(i+1)$-th menu, we directly obtain the foreign keys of the options that the user selected in the 1-st, ..., i-th menus without having to join the dimensions. Opposite to other techniques [LCC+15] that require ad-hoc algorithms inside the DBMS, our approach is based on efficient query plans that are readily available in the query optimizer. For building the $(i+1)$-th menu, the DBMS accesses only the $(i+1)$-th dimension and, depending on the selectivity of the options previously selected by the user, applies one of the following strategies: *i)* if the options previously selected by the users are non-selective (e.g., (s)he selected many feeds), an indexed semi-join is applied: for each entry in the $(i+1)$-th dimension, an index look-up in the fact table using an index on

the $(i + 1)$-th dimension key is done, returning the entry as soon as a measurement satisfying the previous selections is found; *ii)* if the options selected by the user are selective, an indexed hash join is applied: an index on the dimension keys of the fact table is scanned and, for the entries satisfying the conditions on the 1-st, ..., i-th dimension keys, the value of their $(i + 1)$-th dimension key is returned, and a hash join with the $(i+1)$-th dimension is computed. By applying partial evaluation we compute data-driven menus one order of magnitude faster than the state of the art solutions.

For densifying the sparse data cube of the SFDW, we define *derived facts*. For nutritive values that have not been measured in the lab, no record is stored in the DW. Domain experts use chemical formulas between other nutrients for computing values that do not exist in the fact table. Since equijoins retrieve no data if for a given feed sample the nutrients appearing in the formulas have not been measured (which is the case for many feed samples in the sparse data cube of the SFDW), derived facts are computed using Nearest Neighbour Joins (NNJ) [CBB15b]:

1. for each nutrient appearing in the formula a NNJ is applied between the query points and the fact table, and the measurements are retrieved.

2. after all nutrients are processed, the formula defining the derived facts is applied on the NNJ result.

Since formulas (i.e., step 2) can be evaluated efficiently with just a scan of the result, it is important to achieve good performances in computing the NNJ result (i.e., step 1). Standard SQL can be used for computing NNJs [YLK10] but is inefficient since it suffers from index false hits [CBB15b] when additional conditions are specified in the query. This causes the blocks of the fact table to be fetched more than once. Our implementation uses our SQL extension [CBB15b] that includes a Group- and Selection-enabled NNJ operator which is robust if the number of feeds on which a derived nutrient must be computed grows, and if the size of the fact table grows in terms of nutrients measured. Our implementation fetches each block of the fact table at most once (i.e., without redundancies) which is essential for minimizing the response time. We experimentally show the gain in performances we obtain against other query evaluation plans.

Since single dimensions are not selective, computing aggregations over the fact table when only few dimensions are constrained is expensive since many measurements share the same keys. Instead of sampling the data and provide an approximate aggregation value, Feedbase.ch uses materialized views. By using materialized views we compute distributive aggregation functions

over the entire dataset in less than one second. Materialized views pre-aggregate the data at the finest possible level of aggregation allowed to the user, and reduce the number of measurements to query and on which the aggregations must be computed. We compare the use of materialized views for computing distributive aggregation functions against the use of a denormalized fact table and of a star-join.

Our system is used by 3000 Swiss feed mills, research institutions, companies, and private farmers to compose healthy, effective and cheap animal feeding, and to optimize data collection and lab analyzes. We conclude this chapter by describing three use cases of Feedbase.ch.

4.2 Related Work

In this section we describe the lessons learned in implementing the core functionalities of a system querying data from the Swiss Feed Data Warehouse. Systems for assessing the quality of national animal feed are offered by many countries to maximize the quality and minimize the costs of the animal feed production. Our system is the first that allows at the same to explore the spatial, temporal, and biological properties of the data. This is a prerogative for our users, since the nutritive content of the animal feeds strictly depends on the place where the feed has been grown, on the time when it has been harvested, on how mature it was, etc.

The German Feed Database [Ger16] provides historical data from the last 80 years. Just one class of nutrients can be selected per request (e.g., either the Amino Acids, or the Minerals, or the Vitamins, etc.), thus it is impossible to identify correlations between nutrients belonging to different classes. Menus are not data-driven, therefore many null values are presented to the user. For example, when querying the Amino Acids in feed Draff, among the 35 queried nutrients 34 are null. Derived nutrients are computed just on samples for which all needed measurements are available. Aggregated data are presented to the user but no grouping factors can be selected. Thus, it is not possible, for example, to identify regions with high-quality feeds.

The French Feedbase [Fra16] provides, for a given animal feed, the Average value of each nutrient, together with the Min, Max, Count, and Standard Deviation. Nutrients are shown to the user independent if a value exists or not: no data-driven menus are provided to the user for selecting only the nutrients for which data exist. For a given nutrient, a histogram representing the number of measurements with a given value is computed, but the temporal evolution of the nutrient is not given. The correlation between different nutrients cannot be computed either. Derived nutri-

ents are computed only for the samples for which all needed measurements exist. Thus, many null values are presented to the user. The geographic dimension stores at most the country from where a given sample comes, and no possibility of querying specific regions is given. Our system stores the coordinates of the field from which a sample comes, and allows, for example, to find the regions providing the most similar feed to the targeted one.

The American Feed Grains Database [US16] provides on a {yearly, quartile, monthly}-base the price, the quantity, etc., of a given feed produced by each State or imported into the US. The nutritive content of the feeds is not provided, and, apart from a chart representing the temporal evolution of the parameters, no further data processing is offered. Data-driven menus are produced for guiding the user through only meaningful selections but they exhibit poor performances (i.e., several seconds) due to the star join between the dimensions and the fact table. We exploit efficient query plans for building data-driven menus interactively, i.e., in less than a second.

The Asian Feed Ingredient Composition Database [Asi16] provides the nutritive content of Asian feeds. As stated by its authors, the system is not realiable since no geographic information is stored in the system: the nutritive content of the same feed changes dramatically according to place, altitude, etc. where the feed has been grown, and it is important to consider the spatial dimension in the feeding process. The system neither provides any temporal information, nor data-driven menus for querying only selected feeds and nutrients.

In Australia [Aus16], the Feed Analysis Database is available as an offline application. The system supports only windows-based desktop clients, while Feedbase.ch can be used by any client, both using desktop (Windows, Linux, etc.) and mobile devices (Android, Apple, etc.). No data history is available since only the measurements for the current year are available. The database only stores 1250 feed samples and, thus, does not suffer from slow runtime in building the data-driven menus. Only two derived nutrients can be computed, opposite to the 405 of the SFDW, and they are computed only for the feed samples for which all needed measurements are available.

The Sub-Saharan Africa Feed Composition Database [ssa16] provides the values of 23 nutrients for 459 feeds in 14 countries. No temporal dimension is available and only basic aggregations without further (e.g., temporal, spatial, or biological) grouping factors can be computed. The absence of detailed geographical information makes the result not reliable for the same reasons stated above. Data-driven menus are provided. They are fast because of the low amount of data to process (15k feed samples are stored in total in the database) and since they only allow one item

(e.g., one feed) per menu to be selected. Our system does not constrain the amount of selections the user makes and stores millions of records.

In [Kes08] a solution for computing data-driven menus is patented. It is implemented as a *star join* between the fact table and the dimensions. When dealing with historical data this is inefficient (cf. Figure 4.6) since many measurements have the same value for a given foreign key and fetching all of them is expensive. Furthermore, the dimensions are accessed redundantly since for the i-th menu each dimension $\mathbf{D_1}, \ldots, \mathbf{D_i}$ needs to be accessed. Our implementation fetches each dimension only once; it uses the query results of the previous menus for partially evaluating the query for the i-th menu avoiding the join with the dimensions and is scalable. The works in [ER07] and [AR12] introduce interface generators, i.e., solutions for automating the process of building user interfaces using the meta-data of the data warehouse. Both work focus on the design of the user interface (e.g., which and how many textual input fields have to be created) but do not address the problem of driving the user to select only the options for which data exist according to his/her previous selections.

A technique for speeding up joins between the fact table and the dimensions when additional predicates on the dimensions are involved has been proposed by Lahiri et al. [LCC+15] and implemented in the query optimizer of Oracle. It makes a scan of the dimensions, keeps the primary keys of the entries satisfying the predicates, and uses them in the IN condition on the foreign keys of the fact table. Such a condition can be evaluated with just one scan of the fact table. Indices can be used for evaluating the condition on the foreign keys without fetching the actual tuples. Our approach for computing data-driven menus is similar. It uses partial evaluation but, in contrast to the above mentioned approach, does not require ad-hoc algorithms and unlocks efficient query plans already available in the DBMS. We point out two different query plans to adopt depending on the selectivity of the options selected by the user.

The Data Warehouse Toolkit by Kimball [KR13a] introduces *aggregate fact tables* to accelerate the computation of aggregation queries over the fact table. They pre-aggregate the data and store it separately (e.g., as a materialized view) so that the amount of data to query for computing the aggregates decreases. Gupta et al. [GHQ95] introduced an algorithm that, given an input query $\boldsymbol{q}$ computing aggregates over the fact table, returns (if any) a query $\boldsymbol{q}$' over a materialized view computing the same result as $\boldsymbol{q}$. Srivastava et al. [SDJL96] have introduced more general algorithms that enlarge the scope of the query optimizer since they can transform $\boldsymbol{q}$ into the union of several queries $\boldsymbol{q}'_1, \ldots, \boldsymbol{q}'_n$ each on a different materialized view. In this chapter we

measure the gain in performances we obtain in computing nutritive statistics in the SFDW by using materialized views.

The work by Chaudhuri et al. in [CD97] points to several SQL extensions that facilitate and speed up queries over DWs. For Nearest Neighbour Joins (NNJs), standard SQL can be used [YLK10] but its most efficient evaluation plan suffers from index false hits when additional conditions are specified in the query. An SQL extension has been proposed [SAA10] for computing NNJs using SortMerge without index false hits. In the presence of multiple feeds, however, such an extension suffers from redundant fetches, since the *Lateral* subquery [GLJ01] with which different feeds are managed fetches multiple times each block storing measurements of different feeds. Feedbase.ch uses our SQL extension [CBB15b] for computing NNJ queries more efficiently. Our solution allows to compute SortMerge NNJs without index false hits or redundant fetches, and it can take advantage of each optimization of the query optimizer.

4.3 Architecture

In Figure 4.1 we show the architecture of Feedbase.ch. The *Swiss Feed Data Warehouse* (SFDW) module will be presented in Section 4.4 and stores the data. It is extended with the features of the libraries PostGIS [pos16] (including geographic functionalities for modelling the spatial aspects of the data warehouse) and PgNumerics [pgn16] (including mathematical functions for computing similarity tests between feed samples).

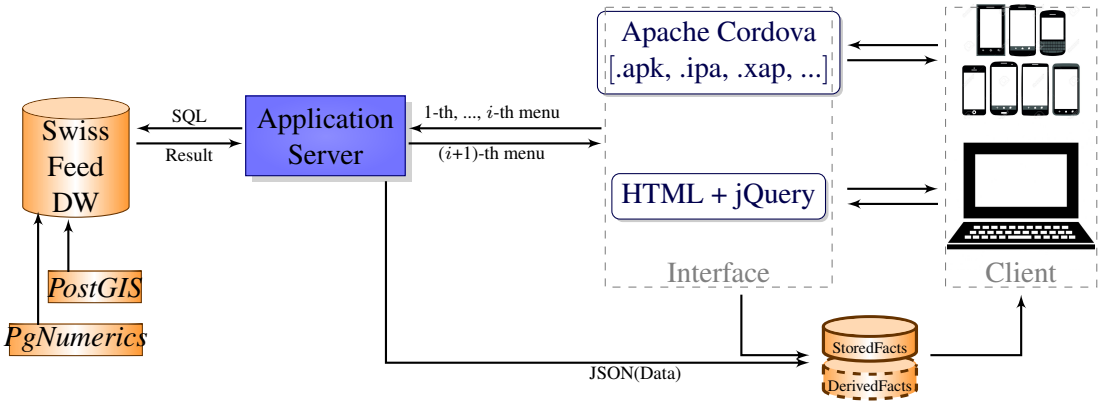


Figure 4.1: Architecture of Feedbase.ch

The *Application Server* module is implemented by Apache, and it sends the queries to the Swiss Feed Data Warehouse, and it receives the result. This module massively interacts with the client

while building the data-driven selection menus, i.e., menus to guide the users to make meaningful selections or, in other words, selections for which data exist in the data warehouse. We will show in the next sections how this process works, and how we ensure good performances. In fact, those menus have to be built instantaneously but require, at the same time, the computation of star-joins. We will show how to use *partial evaluation* for building data-driven menus avoiding redundant joins between the fact table and the dimensions.

Once the user has specified all his/her selections, the Application Server queries the SFDW for retrieving the data. The data result consists of the *stored facts*, i.e., nutritive values that have been physically measured in the lab, and the *derived facts*, i.e., nutritive values that have not been analysed but that are computed based on the value of other nutrients. It is important that stored facts and derived facts share the same schema. This has the advantage that derived facts can be processed with no difference to the stored facts both by the DBMS and by the application. For example, the DBMS can compute any join between the derived facts and the dimension tables since the derived facts have also a value for each dimension key. On the other hand, once the data are forwarded to the client, any data processing activity from the application (such as computing correlations, graphs, statistics, interpolations, geographical distributions) is computed independently whether the facts are derived or stored.

Before forwarding the data to the user, the Application Server converts the result data into a JSON object, i.e., a string storing, for each feed sample, facts (nutrients' measurements) and derived facts (derived nutrients' measurements). As shown in the (simplified) string below, a JSON object can be seen as a table where each *row* stores the data on one given feed sample, and the *cols* specifies the attribute names:

```
{"rows":[
  {"c":[ {#4, Hay, Zug, CH-6300, 2015-05-30, 1.8, ..., 7.1, 0.5, ..., 7.5} ]},
  {"c":[ {#9, Hay, Zug, CH-6300, 2015-06-02, 2.1, ..., 6.4, 0.2, ..., 7.4} ]},
  ],
 "cols":[{SampleID, Feed, Canton, ZIP, Date,
         Nutrient1, ..., Nutrient_n, DerivedNutrient1, ..., DerivedNutrient_n}
]}
```

In our system, returning the data to the client as a JSON object makes the implementation of the user interface simpler since it allows to instantiate any graphical element provided by the Google APIs [Goo16] directly, i.e., without any further data transformation, independent of the graphical element to instantiate. In Feedbase.ch, among the Google APIs we use Google Charts to draw

temporal and correlation charts, Google Maps to represent the geographical distribution of the data, and the Google DataView to show the retrieved data in a table that can be sorted by the user by any parameter.

As shown in Figure 4.1, two graphical *Interface*s of Feedbase.ch are available: one for mobile devices, one for desktop devices. The mobile interface has been designed and implemented using Apache Cordova [cor16]. We have made such a choice since Cordova allows to write a mobile app using HTML and JavaScript, independent of the platform (Android, iOS, Windows Phone, etc.) where the app will be run. The desktop interface has been implemented using standard HTML, and JQuery for triggering event-based queries. As shown in the figure, the *Client* can query Feedbase.ch using any platform. Details on the user interface will be given in Section 4.5.

4.4 The Swiss Feed Data Warehouse

Figure 4.2 shows all key concepts of our data model. The data model underlying the Swiss Feed Data Warehouse is a star-schema where the **FactTable** stores, apart from the measurement's value, only dimension keys. On top of each table in Figure 4.2 we report the number of rows and columns stored in the DW. Each dimensional attribute is replicated 3 times, i.e., once for each different supported language (e.g., english, german, french). Computing translations online using third part services (e.g., Google Translate API) is, in fact, not free of charge and usually is not effective since many biological properties can only be translated by domain experts.

The dimension **Feed** stores the information about 1,300 animal feeds, such as their name (*Pea*), category (*Legume*), botanical name (*Pisum sativum*), and other 15 fields.

The dimension **Sample** stores the information characterizing the 110,000 feed samples that have been analyzed, such as the Lab in which the sample has been analysed (*EuroLab*), how it has been stored (in a *Silo*), how mature it was (*Very mature*), textual information (*Slightly rotten*), and other 76 botanical parameters.

The dimension **Location** stores 3,000 geographical locations from where the feed samples come, such as the city where the feed has been grown (*Zurich*), the postal code (*CH-8046*), the altitude (*400 meters*), the geographical coordinates of the field from where the sample has been extracted (*47°42'/8°50'*), the canton (*Zurich*), etc. Also in such a case, even if from the geographical coordinates every other attribute can be deduced online through third-part services, we store in

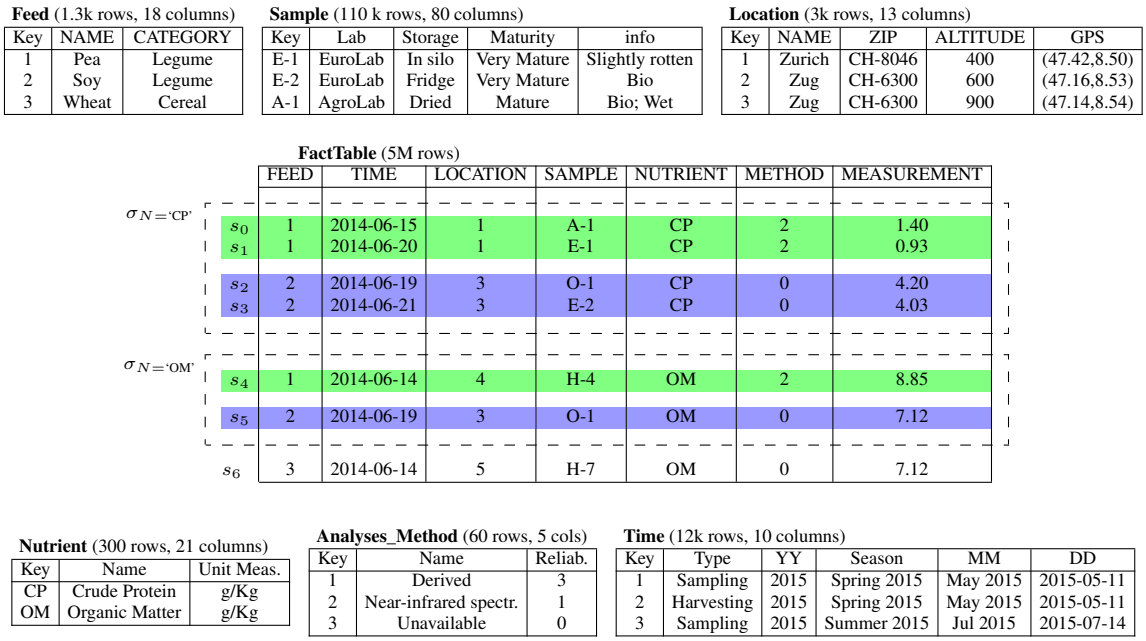


Figure 4.2: Star Schema of the Swiss Feed Data Warehouse.

our data warehouse the entire geographic information. Independent of the selections the user makes (e.g., canton = Zurich), every condition can be directly evaluated in the data warehouse (e.g., for obtaining the canton of a given coordinate). This has two advantages: *i)* each query can be resolved using plain SQL (and thus benefit from all DBMS optimization rules); *ii)* for each location stored in the SFDW, we query third part services only once (i.e., when the location is added to the DW) rather than once for each record returned to the client.

The dimension **Nutrient** stores the information about the 300 nutrients that can be measured on a sample, such as the nutrient name (*Crude Protein*), its abbreviation (*CP*), unit measurement (*g/Kg*), and other 18 columns.

The dimension **AnalysesMethod** stores information about the 60 possible instruments or techniques used for measuring the value of a given nutrient, such as the method name (*Calorimetry*), its reliability (*2*), etc. The attribute Reliability is used when the same nutrient is analysed multiple times on the same sample: in such a case, only the measurement with the higher reliability value is shown to the user.

Finally, the dimension **Time** stores the temporal information in the form of a hierarchy [KR13b] such as year (*2015*), season (*Spring*), month (*May*), etc. More than 12,000 different timestamps are stored in this dimension. As shown in Figure 4.2, the same timestamp may be stored

multiple times in dimension **Time**, but with a different Type value. The attribute Type is a *fact dimension* [KR13c], i.e., it describes which event the timestamp represents. For example, Type='Harvesting' means that a given timestamp describes the moment when the animal feed has been harvested in the field; similarly, analyses timestamp, sampling timestamp, etc. are stored. This design keeps the size of the fact table of the SFDW low since for most measurements the temporal information is incomplete. Furthermore, it is flexible since new temporal information can be added to the warehouse without changing the schema. On the contrary, creating an additional foreign key for a new temporal dimension is inefficient since all measurements previously inserted in the fact table do not have a value for it. This would increase the size of the fact table since for each measurement a dimension key pointing to a "Not Available" entry in the dimension needs to be stored. Specifying a Null foreign key is possible but causes referential integrity violation [KR13d] and can be problematic when writing queries: the schema does not make visible that queries between the fact table and the dimensions need to be modeled through outer joins. The drawback of such a design is that, if for a measurement many timestamps are available, then multiple rows are stored in the fact table (each with a different time key). Such a design is thus not appropriate for dense temporal dimensions: in the fact table of the SFDW, instead, each measurement is only replicated twice on average.

Other tables not directly linked to the Fact Table (such as materialized views or the table storing the formulas defining derived facts) will be discussed in the following sections.

4.5 Interface

Figure 4.3 shows the user interface of Feedbase.ch. It is available at `http://www.feedbase.ch`. It is composed by 4 main areas that communicate with each other, and that are flexible since each area can activate or deactivate different functionalities:

① the menu at the top allows the user to select the Feeds, the Nutrients, the Years, Seasons, the Geographic regions, etc. to query. Since the data cube of the SFDW is sparse (i.e., for many combinations of dimensional values no corresponding data exists in the fact table), Feedbase.ch builds, for each dimension, a *data-driven* menu, i.e., a menu showing only the options for which exist data satisfying the previous user's choices. At the end, the measurements satisfying all selected criterias will be fetched from the fact table and displayed in area ②.

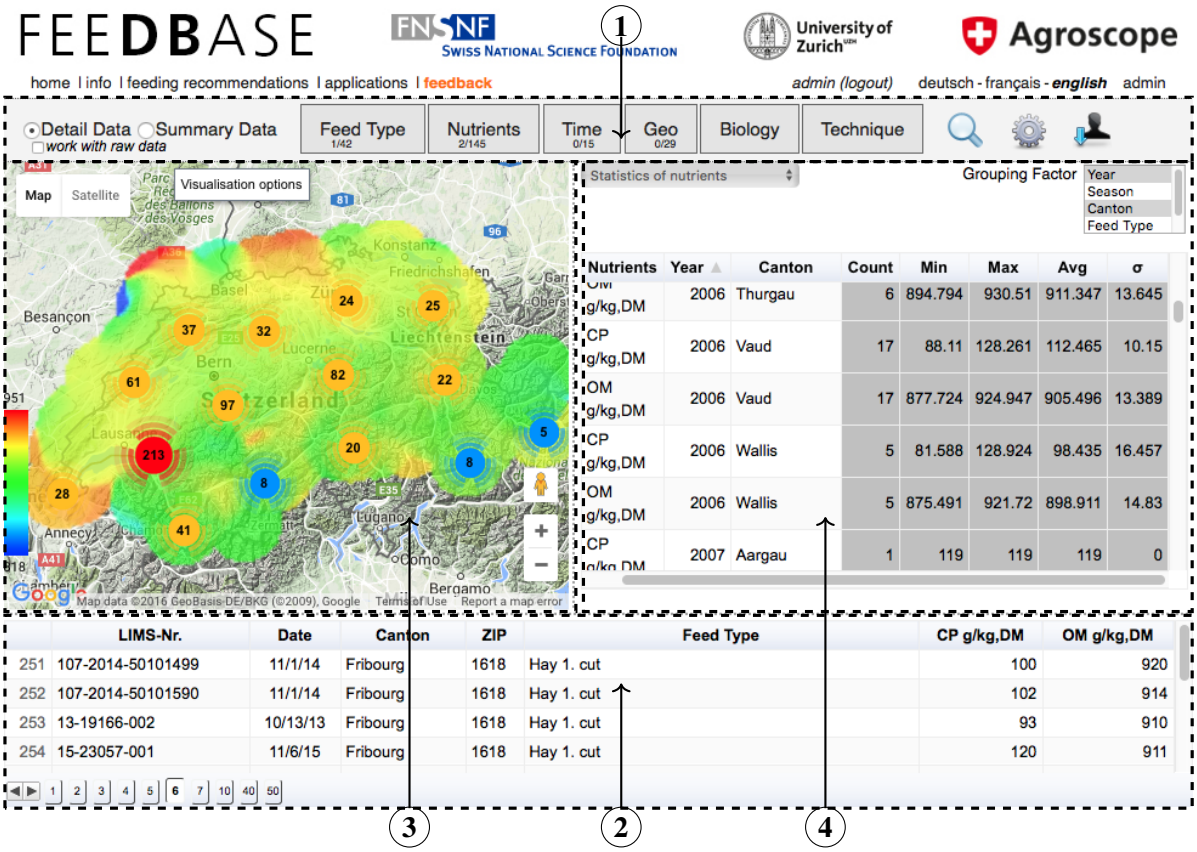


Figure 4.3: interface

② the bottom section provides the data satisfying the user's selections: for each feed sample satisfying the users' selections, the value of the selected nutrients is displayed. For example, in Figure 4.3 the so called Lims Number (i.e., the Sample ID), together with the Time when the sample has been analysed, the Canton and the ZIP code of the location where the feed has been grown, the Feed Type, and the value of the selected nutrients (e.g., Crude Protein *CP* and Organic Matter *OM*) are shown. After a click of the user on one of the result rows, Feedbase.ch will automatically indicate on the map in area ③ the location of the selected sample.

③ the left section provides a *Spatial Representation* of the result. Together with the geographical indication of the origin of the animal feeds, additional information are provided. As the legend of the graph shows, a blue area indicates a region with a low content of a selected nutrient (e.g., Crude Protein), while red areas indicate regions with a high content. This tool is used for identifying the regions where high quality feed (i.e., feed with

high nutritional value) is grown. Markers indicate the regions' density in terms of number of samples analysed. A red marker identifies the regions from where many samples have been extracted (e.g., 213), while blue markers identify regions with few samples (e.g., 5). Domain experts use this tool to identify regions where more analyses should be done.

④ the right section allows to select one of the following four functionalities. Each of them is implemented using standard techniques (such as SortAggregate [Gra93], Student's similarity t-Test, etc.) and will be described within a use case scenario in Section 4.9:

(a) a *Statistic Table* computing aggregations for the selected nutrients (visible in the Figure) over different grouping attributes selected by the user. For computing aggregates on the entire dataset, and not just on the subset presented to the user, without deteriorating performance, we use *materialized views*.

(b) a *Temporal Chart* providing a temporal representation of the nutritive values for finding the nutrients which have experienced different (growth, decrease, instability, etc.) effects over time.

(c) a *Correlation Chart* allowing to discover correlations between the nutrients. Nutrients highly correlated to a given one can be computed as derived nutrients using formulas, rather than measuring them in the lab, to reduce the costs.

(d) the most *Similar Regions*, i.e., the regions with the most similar nutritive content compared to a target region selected by the user. We will show that this functionality is used by farmers who run short of animal feed in their stocks, to buy additional feeds from other regions in Switzerland.

Our interface is different from the other Feed Data Warehouses since it does not provide a unique stand-alone area, but it is composed by multiple areas interacting with each other (e.g., when the user selects a measurement from the Temporal Chart in area ④, then areas ② and ③ automatically show, respectively, the detailed sample information and its geographic location). In other words, our system allows to explore at the same time the spatial, the temporal, and the biological properties of the data.

4.6 Area 1 : Data-Driven Menu

In area ① of the interface, the user selects the feeds, nutrients, etc. for which the measurements must be retrieved. For each menu, a list of selectable options is displayed. However, because of the biological diversity of the animal feeds, many combinations are not compatible with each other, i.e., retrieve no data (e.g., feed Pea is not grown in canton Zurich; nutrient Phosphorus has never been measured for feed Oat Flakes; etc.). For avoiding displaying to the user the many options incompatible with the previous selections (and, thus, returning no data), it is necessary to show only the options for which data exist in the DW according to the previous selections. We now describe the lessons learned in implementing *data-driven menus*.

4.6.1 Star Joins

For computing data-driven menus in a DW system, if m is the number of dimensions in the schema, we need to compute m queries [Kes08] $\boldsymbol{q}_1, \ldots, \boldsymbol{q}_m$, where $\boldsymbol{q}_{i+1}$ returns the next, i.e., the $(i+1)$-th, menu to display:

$\boldsymbol{q}_{i+1}$:

```
SELECT DISTINCT D_{i+1}.value
FROM D_{i+1}
WHERE key IN
      (SELECT FactTable.fkey_{i+1}
       FROM FactTable, D_1, ..., D_i
       WHERE FactTable.fkey_1 = D_1.key AND ... AND FactTable.fkey_i = D_i.key
             AND D_1.value IN (values_1) AND ... AND D_i.value IN (values_i))
```

In query $\boldsymbol{q}_{i+1}$, the result column $\mathbf{D_{i+1}}.value$ returns the options to be displayed in the $(i+1)$-th menu, while $values_1, \ldots, values_i$ in the subquery are the options the user has selected, respectively, in the 1-st,..., i-th menus. As shown in Figure 4.4, for computing $\boldsymbol{q}_1, \ldots, \boldsymbol{q}_m$, a dimension $\mathbf{D_i}$ is joined with the fact table $m - i + 1$ times in total.

(a) State of the art (b) Partial Evaluation

Figure 4.4: Using current solutions, the i-th dimension $\mathbf{D_i}$ is joined to the fact table $m - i$ times.

Example 14. In this example we compute the first three menus of Feedbase.ch applying state of the art solutions. Query $\boldsymbol{q}_1$ computes the Feeds menu by displaying the name of the feeds for which data exists in the fact table.

$\boldsymbol{q}_1$:
```
SELECT DISTINCT Feed.name
FROM Feed
WHERE Feed.key IN
      (SELECT FactTable.feedFkey
       FROM FactTable )
```

Query $\boldsymbol{q}_2$ computes the Nutrients menu by displaying the name of the nutrients measured on the selected feeds (e.g., Hay and Barley).

$\boldsymbol{q}_2$:
```
SELECT DISTINCT Nutrient.name
FROM Nutrient
WHERE Nutrient.key IN
     (SELECT FactTable.nutrientFkey
      FROM FactTable, Feed
      WHERE FactTable.FeedFkey=Feed.key AND Feed.name IN ('Hay','Barley'))
```

Finally, query $\boldsymbol{q}_3$ computes the Time menu by displaying the years during which the selected nutrients have been measured on the selected feeds.

$\boldsymbol{q}_3$:
```
SELECT DISTINCT Time.year
FROM Time
WHERE Time.key IN
     (SELECT FactTable.timeFkey
      FROM FactTable, Feed, Nutrient
      WHERE FactTable.FeedFkey = Feed.key AND
            FactTable.NutrientFkey = Nutrient.key AND
            Feed.name IN ('Hay','Barley') AND
            Nutrient.name IN ('Crude Protein','Vit. A') )
```

The reader can see that, in the IN subuery of $\boldsymbol{q}_1, \ldots, \boldsymbol{q}_m$, the dimensions are redundantly joined to the fact table.

In Figure 4.5 we show the query plan for $\boldsymbol{q}_3$. The plan shows that the **Time** (i.e., the $(i+1)$-th dimension) is scanned and a hash join with the subquery is computed. The subquery is a star-join between the fact table and the **Feed** and **Nutrient** (i.e., the 1-st, ..., i-th) dimensions.

Commercial DBMSs such as PostgreSQL decompose a star-join with i dimensions, as i binary joins (obtaining a right- or left-deep execution tree [CMX13]). Thus, as shown in Figure 4.5, first the join between the **FactTable** and the **Feed** dimension is computed, then the one with the **Nutrient** dimension. Although selection push-down is applied to the dimensions before the star-join, such a deep query execution tree is not efficient since single dimensions are not selective. Thus, many measurements are retrieved and fetched from the fact table by the first join.

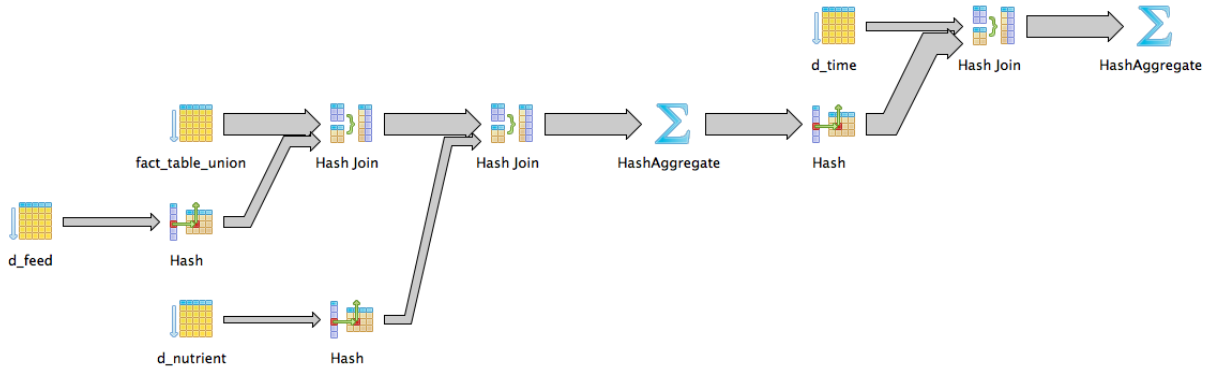


Figure 4.5: Building of the Time menu in the SFDW using Hashing.

In Figure 4.6 we show that the previous query plan is not suitable for building data-driven menus in the SFDW since it takes around 5 seconds to complete, i.e., it does not provide interactive performances to the user. This is a problem in our system since for each request in Feedbase.ch the user computes many data-driven menus (each resulting in a new star-join): for example if (s)he is interested in the nutritive content of the Cereals during Summer in the Northern cantons of Switzerland then (s)he will make selections on 4 menus (feeds, nutrients, seasons, and cantons) causing a total response time of 20 seconds. In Figure 4.6 we also show that although an index can be used to access only selected tuples of the fact table (and avoid hashing the entire fact table) it does not reduce the response time. This is so because single dimension keys are not selective in DWs (e.g., many tuples exist with the same feed value), and many tuples are returned by an indexed join (e.g., the first). Using a composite index slightly increases the performances since it allows to compute all i joins without fetching any tuple from the fact table (all foreign keys are stored in the index). However the query would still require 3 seconds to complete (thus, 12 seconds waiting time for 4 menus) since many keys are returned by the first join. In the figure we also show the runtime of using a denormalized fact table. It consists in storing in the fact table also the dimensional attributes, so that each condition involving them can be evaluated without joining the fact table with the dimensions. However the size of each tuple grows with

one order of magnitude making a scan of the fact table half a minute. Indexes help to fetch only selected tuples from the denormalized fact table but, because the index is built on the dimensional attributes values (and not on the dimension keys), it takes several seconds to answer the query.

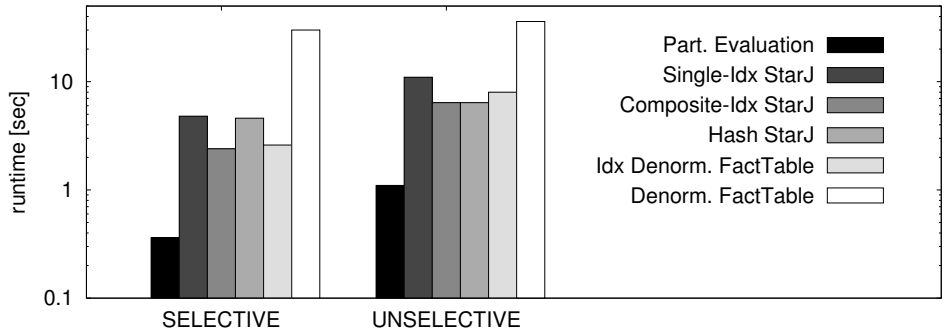


Figure 4.6: Building of the Time menu in the SFDW.

Summarizing, for $\boldsymbol{q}_1, \ldots, \boldsymbol{q}_m$, two drawbacks need to be addressed for reducing the response time of the user:

1. a dimension $\mathbf{D_i}$ is in total joined to the fact table $m - i + 1$ times
2. the join with $\mathbf{D_1}$ returns many false positives, i.e., tuples that do not satisfy the predicates on $\mathbf{D_2}$ or $\mathbf{D_3}$, etc.

We will now show how recently proposed techniques can be used to address the second drawback above; then we will introduce partial evaluation for also ensuring that each dimension is not accessed more than once.

4.6.2 Single Scan

Techniques for enhancing the performances of the join between the fact table and the dimensions have been introduced [PP15; LR98] and implemented in Oracle [LCC+15]. They do not transform the join with m dimensions into m binary joins, and avoid the many false positives returned by singular joins. Given query $\boldsymbol{q}_{i+1}$, they decompose its evaluation into multiple blocks:

1. for each involved dimension $\mathbf{D_i}$:
   1.1 apply push down selection to $\mathbf{D_i}$

1.2 read and keep the primary key of the tuples satisfying the predicate

2. scan the fact table and return the tuples satisfying an IN predicate on the foreign keys obtained in step 1.

As an advantage of such an approach, a tuple is returned if it satisfies the entire WHERE clause, and no false positive are returned. The disadvantages are that: *i)* the kernel of the DBMS needs to be extended (commercial DBMSs do not decompose a given query evaluation tree into multiple trees); *ii)* a dimension $\mathbf{D_i}$ is in total still accessed $m - i + 1$ times: this cost becomes relevant in the presence of high dimensional data (m is the number of dimensions).

4.6.3 Partial Evaluation

While the previous approach requires new algorithms to be implemented inside the core of the DBMS, our approach unlocks efficient query plans already existing in current DBMS implementations. It is this supported by any commercial DBMS implementation and also avoids to redundantly access $\mathbf{D_1}, \ldots, \mathbf{D_i}$.

Our approach performs $\boldsymbol{q}_{i+1}^{part.}$ (computing the $(i+1)$-th menu) using *partial evaluation*. For evaluating $\boldsymbol{q}_{i+1}^{part.}$, among the dimension tables, we only accesses $\mathbf{D_{i+1}}$ (this is necessary since we need to add the dimension's values to the new menu) and use the results of $\boldsymbol{q}_1^{part.}, \ldots, \boldsymbol{q}_i^{part.}$ for avoiding joining the fact table with the dimensions. For $\boldsymbol{q}_{i+1}^{part.}$ we take advantage at the same time of our system Feedbase.ch (which stores the result of $\boldsymbol{q}_1^{part.}, \ldots, \boldsymbol{q}_i^{part.}$) and of the DBMS (which answers the query). As shown in the SQL statement below, $\boldsymbol{q}_{i+1}^{part.}$ uses directly the keys (i.e., $keys_1, \ldots, keys_i$) of the options selected by the user in the 1-st, ..., i-th menu:

$\boldsymbol{q}_{i+1}^{part.}$:

```
SELECT Key, Value
FROM D_{i+1}
WHERE Key IN (SELECT [DISTINCT] fkey_{i+1}
              FROM FactTable
              WHERE fkey_1 IN (keys_1) AND ... AND fkey_i IN (keys_i))
```

Note that $\boldsymbol{q}_{i+1}^{part.}$ does not anymore join dimensions $\mathbf{D_1}, \ldots, \mathbf{D_i}$ since the keys of the selected options have been retrieved, respectively, in $\boldsymbol{q}_1^{part.}, \ldots, \boldsymbol{q}_i^{part.}$, and temporarily stored on Feedbase.ch.

In Figure 4.7, we show the steps that we make in our system for constructing $\boldsymbol{q}_{i+1}^{part.}$. First, the Client sends to the Application Server the $keys$ of options selected in the 1, ..., i-th menus. For example, if the selected Feeds are 'Hay' and 'Soy', and the selected Nutrients are 'Crude Protein' and 'Vitamin D', then $keys_1 = \{1, 2\}$ and $keys_2 = \{$'CP', 'Vit. D'$\}$. Afterwards, the Application Server instantiates $\boldsymbol{q}_{i+1}^{part.}$ and sends the query to the SFDW. The answer is used for populating the $(i+1)$-th menu.

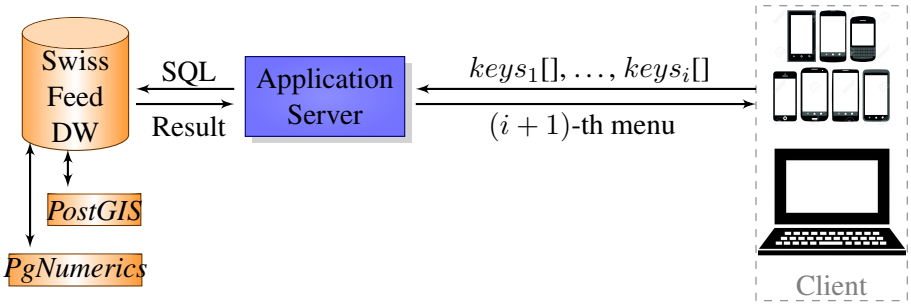


Figure 4.7: Steps in creating the $i+1$-th data-driven menu.

Two strategies for evaluating $\boldsymbol{q}_{i+1}^{part.}$ are possible depending on the selectivity of the selected options:

1. if they are non-selective (e.g., the user selected many feeds), an indexed semi-join is applied. For each entry in the $(i+1)$-th dimension, an index look-up with the key of the entry is done in the fact table and, as soon as a measurement satisfying the previous selections is found, we stop fetching tuples from the fact table and process the next entry in the dimension. This plan is shown in Figure 4.8: for each entry in the dimension $\mathbf{Time}$, an index look-up with $\mathbf{FactTable}.TimeFkey = \mathbf{Time}.key$ is done in the fact table and, as soon as a measurement for one of the selected feeds and nutrients is fetched, the entry is returned and displayed in the Time menu.

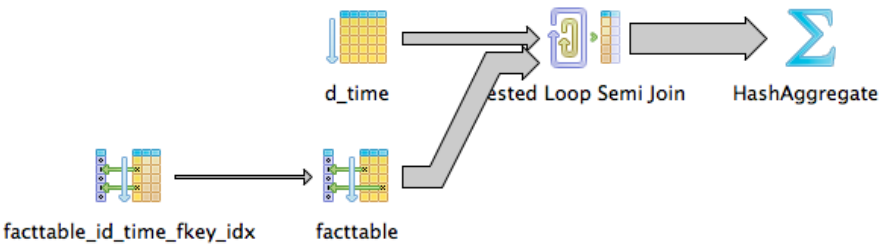


Figure 4.8: Building of the Time menu in the SFDW for non-selective options.

2. if they are selective, an indexed hash join is applied. An index on the foreign keys of the fact table is scanned and, for the entries whose 1-st, ..., $i-$th foreign keys satisfy the WHERE clause of the query, we keep their foreign key to the $(i+1)$-th dimension. We then apply a hash join between those key values and the $(i+1)$-th dimension. This plan is shown in Figure 4.9: first the index on the foreign keys $(FeedFkey, NutrientFkey, TimeFkey)$ is scanned and, for the entries having one of the selected $FeedFkey$ and $NutrientFkey$ values, the value of $TimeFkey$ is returned; then a Hash Join between the dimension **Time** and the returned $TimeFkey$ values is computed. In the specific plan the reader can see that (through a HashAggregate) the DBMS removes the duplicates among the $TimeFkey$ values so that the hash table can be kept in memory.

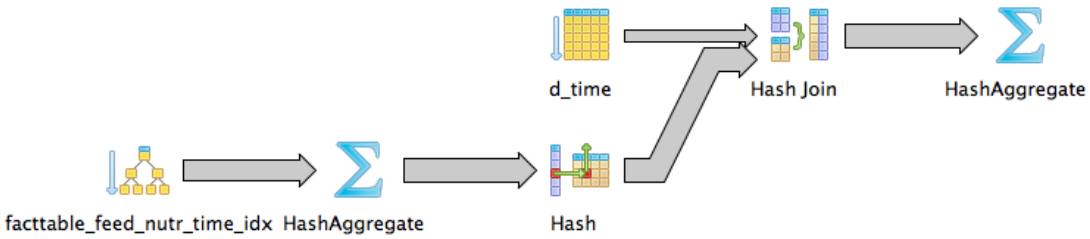


Figure 4.9: Building of the Time menu in the SFDW for selective options.

For an efficient evaluation of the above query plans it is important that an index on the foreign keys of the fact table is available. It allows to directly evaluate the WHERE condition of the subquery without fetching any tuple and avoids to scan the entire fact table, which is big and takes seconds.

Example 15. Consider the data-driven menu *Nutrients* in the Area ① of Figure 4.3. The array $keys_1$ stores the keys of the selected *Feeds*. For example, if the users selects feeds Pea and Soy, then the Application Server receives $key_1 = \{1, 2\}$. The query we use to build the *Nutrients* menu is:

```
SELECT Key, Name
FROM Nutrient
WHERE Key IN (SELECT DISTINCT nutrient_fkey
              FROM FactTable
              WHERE feed_fkey IN (1,2) )
```

In Figure 4.6 we show that in the SFDW partial evaluation is the most competitive approach in terms of runtime for creating data-driven menus. The approach is robust both for a selective

subquery (e.g., only the Roughage as feeds, and the Minerals as nutrients), and for a non-selective one (e.g. all feeds and nutrients have been selected). The figure also shows that although a data driven menu can be computed directly (as a selection) on a denormalized fact table using a composite index, the runtime is an order of magnitude higher than the one for partial evaluation since the index on the dimensional values (here stored in the fact table) is 5 times larger than an index on the dimension keys.

4.7 Area 2: Stored And Derived Facts

In Figure 4.10 we show that once the user has made the selections in Area ①, the Client sends the $keys$ of the selections to the Application Server. The Application Server then builds an SQL query that retrieves from the SFDW the data to return to the user. The SFDW stores data on a version of PostgreSQL that we extended with Nearest Neighbour Joins with robust support of groups and predicates [CBB15b]. As shown in the following example query, the data is retrieved by the union of two queries: the first fetches the stored facts (i.e., the measurements from the fact table that satisfy the users selections), and the second computes the *derived facts*. Derived facts densify the sparse data cube of the SFDW by computing nutrients that have not been physically measured (in the lab), and that are therefore derived using chemical formulas.

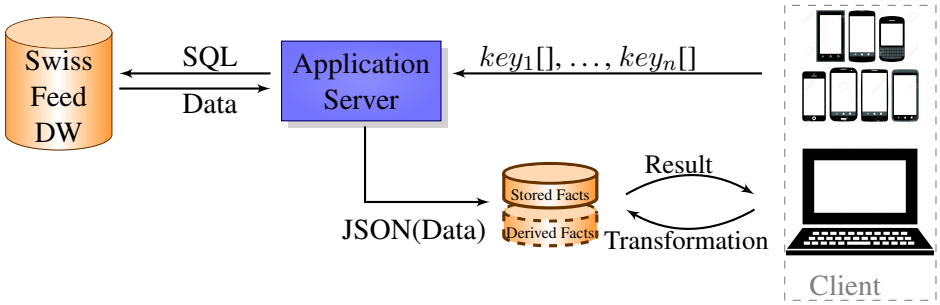


Figure 4.10: Processing of the result: once the user specifies the restrictions on the dimensions, stored facts and derived facts are retrieved and returned as a JSON string.

Example 16. In this example query, we retrieve all Vitamin A and Calcium (having keys 'Vit. A' and 'Ca') measurements on the samples of feeds Pea and Soy (with keys 1 and 2) grown in Zug (locations with key 2, 3); for the same samples, we compute the derived nutrient 'GE'.

```
SELECT Feed, Time, Location, Sample, Nutrient, Method, Measurement
```

```
 FROM FactTable
 WHERE FeedKey IN (1,2) AND
       NutrientKey IN ('Vit. A','Ca') AND
       LocationKey IN (2,3)
UNION
(WITH Z0 AS (SELECT Feed, Time, Location, Sample
             FROM FactTable
             WHERE FeedKey IN (1,2) AND LocationKey IN (2,3))
 SELECT Feed, Time, Location, Sample, 'GE', 1, 0.8 * CP +2* OM
 FROM (SELECT Z0.*, FactTable.Measurement AS OM
       FROM (SELECT  Z0.*, FactTable.Measurement AS CP
             FROM  Z0 NNJ FactTable ON TIME USING FEED
             WHERE NutrientKey='CP' AND LocationKey IN (2,3)) AS Z1
             NNJ FactTable ON T USING G
       WHERE NutrientKey = 'OM' AND LocationKey IN (2,3)) AS Z2)
```

4.7.1 Derived Nutrients' Computation

For calculating nutrients that have not been measured on a feed sample, formulas are used. Table **Formulas** in Figure 4.11 records the formulas for calculating *derived facts*. Attribute D is the derived nutrient, while F stores the formula $f(n_1, ..., n_p)$ for calculating it, where $n_1, \ldots, n_p$ are nutrients. For instance, in table **Formulas**, derived nutrient 'GE' (*Gross Energy*) is defined by the formula $CP * 0.8 + OM * 2$ where 'CP' is the value of nutrient *Crude Protein*, and 'OM' is the value of nutrient *Organic Matter*. In the Swiss Feed Data Warehouse 405 different derived nutrients exist. Since, for a given feed sample, some of the nutrients $n_1, \ldots n_p$ might also not have been measured (remember that the data cube of the SFDW is sparse), computing derived nutrients using equijoins returns an empty result for most of the queries run through Feedbase.ch. Derived nutrients are therefore computed using *nearest neighbour joins* (NNJs) [CBB15b], e.g., using the temporally closest measurements available for $n_1, \ldots, n_p$ on the same feed. We will show that implementing NNJs with standard SQL is possible [YLK10] but expensive when the number of tuples grows. We therefore use our SQL extension that integrates a group- and selection-enabled NNJ and that is proved to fetch each block of the fact table at most once.

Definition 4. *(NNJ).* Assume relations $\boldsymbol{R}$ and $\boldsymbol{S}$ whose schemas include a grouping attribute G and a similarity attribute T. A group- and selection-enabled Nearest Neighbour Join query

Formulas

	D	F
$\rightarrow d_0$	GE	0.8 * CP + 2 * OM
d_1	AP	TS * 0.8 * (0.1 * (OM + CP * 0.9 - CF - FP/2) - CP * 0.9)
d_2	DP	49 + (144 * CP / OM) - 2 * (CP / OM)^2
d_3	AME	7690 - (7.69 * RA) + (6.464 * RP) + (29.43 * RL) - (16.09 * RF)

Figure 4.11: Table **Formulas** Storing Derived Nutrients' Definitions

(SELECT $\star$ FROM $\boldsymbol{R}$ NNJ $\boldsymbol{S}$ ON T USING G WHERE θ) returns, for a given tuple $r \in \boldsymbol{R}$, the tuple(s) $s \in \boldsymbol{S}$ that satisfy condition θ, and have the same group G and the closest T value.

Example 17. Consider relation $\boldsymbol{Z}_0$ in Figure 4.12 representing the samples of feeds 1 (i.e., Pea) and 2 (i.e., Soy) of the Fact Table, i.e.:

$\boldsymbol{Z}_0$:
```
SELECT DISTINCT FEED, TIME, LOCATION, SAMPLE
FROM FactTable AS Z0
WHERE FEED IN (1,2)
```

$\boldsymbol{Z}_0$	FEED	TIME	LOCATION	SAMPLE
z_0	1	2014-06-14	4	H-4
z_1	1	2014-06-15	1	A-1
z_2	1	2014-06-20	1	E-1
z_3	2	2014-06-19	3	O-1
z_4	2	2014-06-19	3	O-1
z_5	2	2014-06-21	3	E-2

$\boldsymbol{Z}_2$	FEED	TIME	LOCATION	SAMPLE	CP	OM
$z_0 s_0 s_4$	1	2014-06-14	4	H-4	1.40	8.85
$z_1 s_0 s_4$	1	2014-06-15	1	A-1	1.40	8.85
$z_2 s_1 s_4$	1	2014-06-20	1	E-1	0.93	8.85
$z_3 s_2 s_5$	2	2014-06-19	3	O-1	4.20	7.12
$z_4 s_2 s_5$	2	2014-06-19	3	O-1	4.20	7.12
$z_5 s_3 s_5$	2	2014-06-21	3	E-2	4.03	7.12

Figure 4.12: First NNJ: The CP value of the tuples in $\boldsymbol{R}$ is retrieved.

Our goal is obtaining table $\boldsymbol{Z}_2$ in Figure 4.12, i.e., we want to find for each of the samples in $\boldsymbol{Z}_0$, the closest 'CP' and the closest 'OM' measurements available for the same feed. Thus, first we compute a NNJ between $\boldsymbol{Z}_0$ and the 'CP' measurements of the fact table, and then a NNJ with the 'OM' measurements. This is done in the following queries where $\boldsymbol{Z}_1$ indicates the result of the first NNJ:

$\boldsymbol{Z}_1$:
```
SELECT  Z0.*, FactTable.MEASUREMENT AS CP
FROM  Z0 NNJ FactTable ON TIME USING FEED
WHERE NUTRIENT='CP'
```

$\boldsymbol{Z}_2$:
```
SELECT Z1.*, FactTable.MEASUREMENT AS OM
FROM  Z1 NNJ FactTable ON TIME USING FEED
WHERE NUTRIENT='OM'
```

The result $\boldsymbol{Z}_2$ of this query is shown in Figure 4.12. Consider the set of 'CP' measurements (delimited in the Fact Table of Figure 4.2 by the first dashed subset) and the 'OM' ones (delimited by the second dashed subset). Tuple $z_0 \in \boldsymbol{Z}_0$, is joined with fact s_0 i.e., its closest CP measurement available, and with fact s_4, i.e., its closest OM measurement available. Note that tuple s_6, although it is the closest OM measurement available for r_0, has not been chosen as nearest neighbour since it belongs to a different group (feed). On top of a NNJ result we compute the derived facts.

Definition 5. *(Derived Facts).* Let $\boldsymbol{Z}_0$ be a set of query points and $\boldsymbol{S}$ a fact table, both with a grouping attribute G and a similarity attribute T. Given a derived nutrient dn and its formula $f(n_1, ..., n_p)$, the *derived facts* $\boldsymbol{q}$ are defined as follows:

$$\mathbf{Z_i}: \quad \texttt{SELECT } \boldsymbol{Z}_{i-1}\texttt{.*, } \mathbf{FactTable}.M \texttt{ AS n}_i \;\; \texttt{FROM } \boldsymbol{Z}_{i-1} \texttt{ NNJ } \mathbf{FactTable} \texttt{ ON } T \texttt{ USING } G, \;\; \texttt{WHERE } \mathbf{FactTable}.N = \texttt{'}n_i\texttt{'} \qquad 1 \leq i \leq p$$

$$\boldsymbol{q}: \quad \texttt{SELECT } \boldsymbol{Z}_0\texttt{.*, 'dn', } f(n_1, ..., n_p) \texttt{ FROM } \boldsymbol{Z}_p$$

where N stores the name of the measured nutrient, M its measurement, and n_i is the i-th nutrient appearing in the formula $f(n_1, ..., n_p)$.

Given a formula $dn = f(n_1, ..., n_p)$, the derived nutrient dn is computed by applying a NNJ for each nutrient n_i appearing in the formula. The first part of the definition above computes a sequence of NNJs where the i-th NNJ fetches from the Fact Table the value of nutrient n_i. Once a NNJ for all p nutrients appearing in the formula has been computed, the second part of the definition calculates the derived facts evaluating the formula on $\boldsymbol{Z}_p$.

Example 18. We use Definition 2 for computing the derived facts $\boldsymbol{q}$ in Figure 4.13, i.e., computing a value for the derived nutrient 'GE'. From table **Formulas**, we get 'GE' $= f($'CP', 'OM'$) = 0.8 * CP + 2 * OM$: a sequence of two NNJs, one fetching the 'CP' values and one the 'OM' values, needs to be computed. We use the join sequence $\boldsymbol{Z}_2$ computed in the previous example, and calculate the derived facts as:

$\mathbf{q} =$ `SELECT FEED, TIME, LOCATION, SAMPLE, 'GE', 1, 0.8*CP + 2*OM FROM` $\boldsymbol{Z}_2$

Consider $\boldsymbol{Z}_2$ in Figure 4.12: the formula $0.8 *$ 'CP' $+ 2 *$ 'OM' calculates the value of the derived facts and produces the result shown in Figure 4.13.

q	FEED	TIME	LOCATION	SAMPLE	NUTRIENT	METHOD	MEASUREMENT
$r_0 s_0 s_4$	1	2014-06-14	4	H-4	GE	1	18.8
$r_1 s_0 s_4$	1	2014-06-15	1	A-1	GE	1	18.8
$r_2 s_1 s_4$	1	2014-06-20	1	E-1	GE	1	19.0
$r_3 s_2 s_5$	2	2014-06-19	3	O-1	GE	1	20.1
$r_4 s_2 s_5$	2	2014-06-19	3	O-1	GE	1	20.1
$r_5 s_3 s_5$	2	2014-06-21	3	E-2	GE	1	19.2

Figure 4.13: Derived Nutrient 'GE' is Computed from Nutrient 'CP' and Nutrient 'OM'.

As shown in Figure 4.13, the schema of the obtained result is identical to the schema of the **FactTable**: the query result represents an extension of the fact table, with derived facts.

In Figure 4.14 we show the evaluation plan for the previous query. For the first NNJ two indexed selections are done on the fact table: the first fetches $\boldsymbol{Z}_0$, the second its 'CP' measurements . The returned tuples are then sorted and the NNJ is computed using SortMerge. The 'OM' measurements are then fetched and the second NNJ is computed similarly.

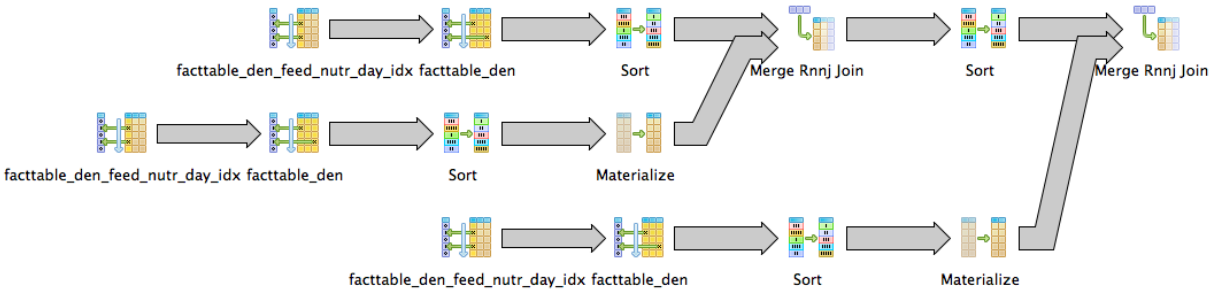


Figure 4.14: Derived Facts using a Group- and Selection-Enabled SortMerge.

In Figure 4.15 we show the performances for computing the derived nutrient DOM (Digestible Organic Matter). It is computed by a sequence of two NNJs. We compare three different query evaluation plans: the first uses an Indexed Group- and Selection-Enabled SortMerge [CBB15b]; the second uses an Indexed Lateral SortMerge [GLJ01]; the third uses a B-Tree [YLK10]. The Indexed Group- and Selection-Enabled Sort-Merge (whose plan is shown in Figure 4.14) is the fastest approach since it does not compute redundancies: for each NNJ, it fetches at once all needed tuples, sorts them, and computes the joins in a single scan of the tuples. The Lateral Sort-Merge is slower since it decomposes a NNJ into multiple NNJs, i.e., one per feed. Thus, blocks storing tuples of different feeds are refetched multiple times. It is important to avoid redundancies for minimizing the response time to the user. The query plan based on the B-Tree is illustrated in Figure 4.16. For each tuple, two index look ups are done for computing a (e.g., the first) NNJ: the first fetches its predecessor, the second its successor. The closest between the two retrieved tuples is the nearest neighbour. Similarly the second NNJ is computed. Such an

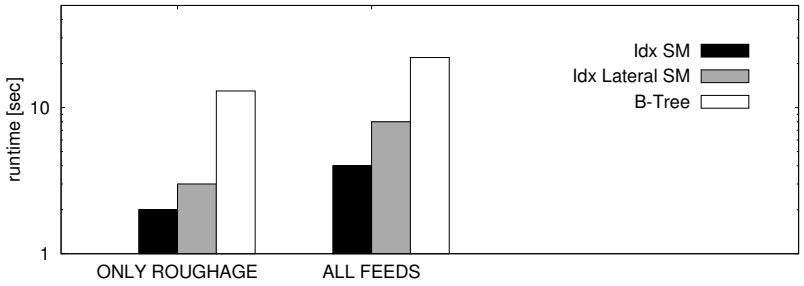


Figure 4.15: Computation of derived nutrient DOM (Digestibility of Organic Matter), through a sequence of two NNJs.

approach inherits the issues of an index-join: it is efficient only if a small number of look-ups are computed. In Figure 4.15 we have first used 30k output tuples (for computing DOM on Roughage), and then 60k outer tuples (for computing DOM on all feeds).

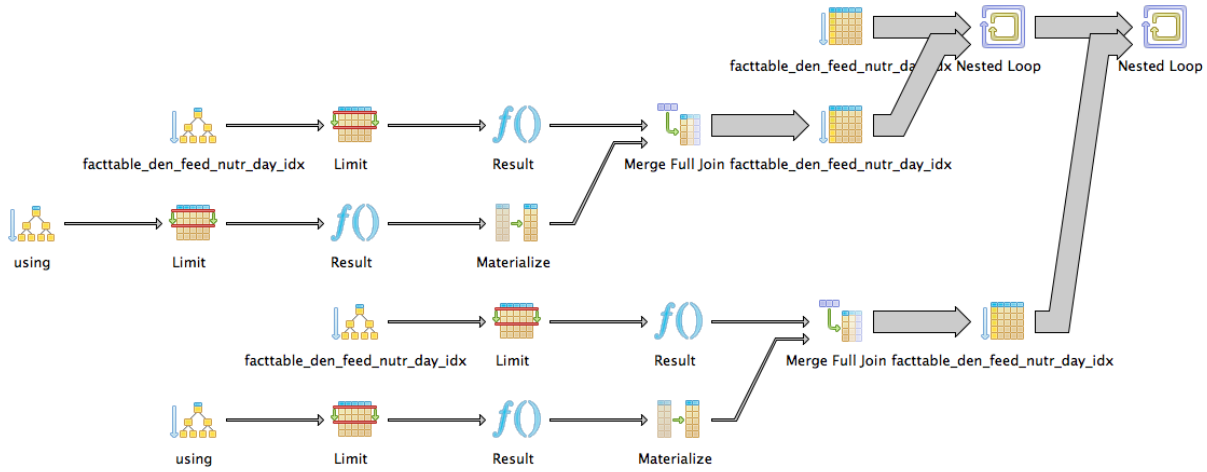


Figure 4.16: Derived Facts using a B-Tree.

4.8 Area 4: Efficient Statistics

Aggregates are used in Feedbase.ch to compute statistics about the nutritive content of the animal feeds queried by the user. For each value of n selected grouping factors (i.e., dimensional values), the statistics are calculated through m aggregation functions, i.e.,

$$GROUP_1,...,GROUPn\,\vartheta\,AGG_1,...,AGGm \tag{4.1}$$

For example, in the area ④ of Figure 4.3, we compute $_{NUTRIENT,YEAR,CANTON}\vartheta_{COUNT,MIN,MAX,AVG,ST\_DEV}$ on the returned measurements: for each of the selected Nutrients, Years, and Cantons, the *count* of the number of measurements, the *minimum, maximum, average value* and *standard deviation* are returned. This offers to domain experts the possibility to discover how the nutritive content changes in different Swiss Cantons; in which Canton a given nutrient has not been measured frequently enough (e.g., CP has been measured only once in Canton Aargau during 2007); where and when a nutrient has a high variability, and where and when it is more stable; etc.

Since in a DW single dimensions are not selective, computing aggregations over the fact table when only few dimensions are constrained by the user is expensive because many measurements share the same keys. For computing statistics that are valid over the entire dataset (and not just on the sample presented to the users) efficiently, Feedbase.ch uses *materialized views*. Materialized views store partial aggregates, and are then used as a base-line for computing online the required aggregates. Materialized views reduce the amount of data on which the aggregates must be computed, and speed up the computation. The materialized view **PartialAggregates** in Figure 4.17 stores the distributive aggregates (Count, Min, Max, Sum) computed over the fact table of the SFDW for each value of the finest possible grouping factor.

PartialAggregates

G_1	G_2	G_3	G_4	G_5	G_6	G_7	F_1	F_2	F_3	F_4
Nutrient	Feed	Maturity	Canton	AltitudeClass	Year	Trimester	min	max	$count$	sum
CP	Hay	1	Aargau	Low	2010	1	1	2	10	1.3
CP	Hay	1	Aargau	Low	2010	…	…	…	…	…
CP	Hay	1	Aargau	Low	2010	4	1	2	10	19.3
CP	Hay	1	Aargau	Low	2011	1	1	2	10	1.3
CP	Hay	1	Aargau	Low	…	…		…	…	…
CP	Hay	1	Aargau	Low	2015	4	1	2	10	1.3
CP	Hay	1	Aargau	Med	2010	1	1	2	10	1.3
CP	Hay	1	Aargau	…	…	…	…	…	…	…
CP	Hay	1	Aargau	High	2015	4	1	2	10	1.3
CP	Hay	1	Appenzell A.	Low	2010	1	1	2	10	1.3
CP	Hay	1	…	…	…	…	…	…	…	…
CP	Hay	1	Zurich	High	2015	4	1	2	10	1.3
CP	Hay	2	Aargau	Low	2015	4	1	2	10	1.3
…	…	...	...	...	...	...	...	...	...	…

Figure 4.17: Materialized Partial Aggregates

In order to reduce the amount of data to store (and then fetch) on the materialized view, Feedbase.ch limits the grouping factors to the following dimensions: (Nutrient, Feed, Stage of Maturity) as biological groups, (Canton, AltitudeClass) as geographic groups, and (Year, Trimester) as temporal groups. Thus all tuples within each of such groups, are aggregated together offline and stored in the materialized view. For example, the first row of **PartialAggregates** stores the minimum, the maximum, the count, and the sum of all Crude Protein measurements done on

feed Hay at the 1st stage of maturity harvested in canton Aargau during the first Trimester of year 2010 at a Low altitude (i.e., $< 600m$). As shown in the query plan of Figure 4.18, when the user asks for statistics we apply standard aggregation techniques (e.g., SortAggregate) directly to **Partial Aggregates**.

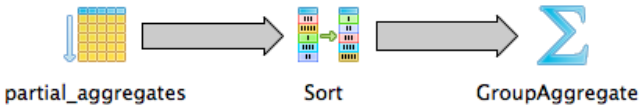


Figure 4.18: Statistics on the materialized view **Partial_Aggregates**.

Example 19. In this example we compute the statistics shown in the area ③ of Figure 4.3, i.e., for feed Hay the content of nutrients CP and OM for each Year and Canton. First the tuples of feed Hay and nutrients CP or OM are fetched from **Partial Aggregates**. Then, by applying SortAggregate or HashAggregate, the tuples having the same value for Nutrient, Year, and Canton are aggregated. For a given group, we compute the COUNT as the sum of the counts, MIN as the minimum of the mins, MAX as maximum of the maxs, SUM as the sum of the sums, AVERAGE as the divisions between SUM and COUNT.

```
SELECT NUTRIENT, YEAR, CANTON, MIN(min), MAX(max), SUM(count),
          SUM(sum), SUM(sum)/SUM(count) AS avg, STDDEV(sum/count)
FROM PartialAggregates
WHERE Feed='Hay' AND Nutrient  IN ('CP','OM)
GROUP BY Nutrient, Year, Canton
```

As shown in Figure 4.19, by using materialized views, Feedbase.ch allows to compute statistics over the entire history of the 18 most measured Nutrients within one second. All distributive aggregation functions computed are correct and accurate. ST_DEV, which is non-distributive, requires every individual measurement in order to be precise. It is estimated in our system as the standard deviation of the averages. If the user wants to compute precise standard deviations, Feedbase.ch offers the possibility of computing statistics using the individual measurements. This consists of a StarJoin between the fact table and the dimensions. As shown in Figure 4.19, this increases the runtime by more than one order of magnitude. Using a denormalized fact table slightly speeds up performances since no join needs to be computed. However, since a denormalized fact table stores single measurements, computing aggregates is slower than using Materialized Views because many measurements need to be fetched.

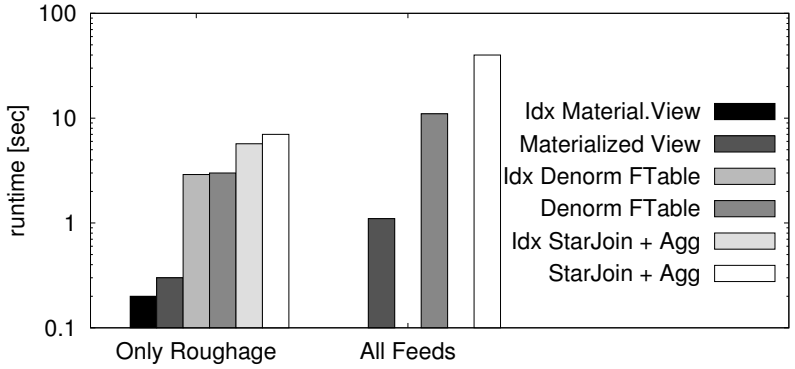


Figure 4.19: Aggregation in the SFDW providing, for each Swiss canton, the statistics of 18 Nutrients (Basic, Carbohydrates, and Minerals).

4.9 Use Cases

Feedbase.ch is used by 3000 Swiss feed mills, research institutions, companies, and private farmers to compose healthy, effective and cheap animal feeding, and to optimize data collection and lab analyzes. In this section, we describe three use cases of Feedbase.ch.

4.9.1 Summer Drought (Year 2015)

From early June 2015 to end of August 2015, lacking precipitation and heat waves in Switzerland almost stopped any growth in pastures and meadows over several weeks resulting in forage shortage. On one hand, farmers were forced to supplement their cows at pasture with hay of the first cut (stored for the upcoming winter) and on the other hand, they were confronted with low hay yields of the second and third cut. At the beginning of the winter feeding period, some farmers had to restock their hay supply from other regions. Farmer X was able to purchase hay from central Switzerland. He would like to have an indication on the hay quality, but he does not have the financial means for paying chemical analyses. Based on the zip code (CH-6017, corresponding to town Ruswil) of the origin of the feed, the farmer uses Feedbase.ch to come to an estimation of the quality of the hay he bought. He keys in the geographical information he knows (e.g., the zip) and he specifies the radius of tolerance: the measurements on hay grown within such a radius during year 2015 will be considered. Figure 4.20 shows the result gotten with a tolerance of 5 km. On the left side we show the places where the analyzed samples come from. On the right we show statistics on their nutritive values.

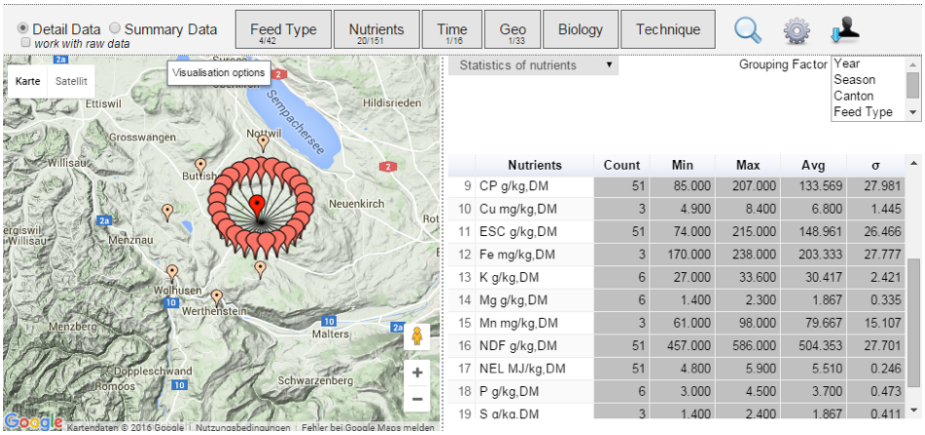


Figure 4.20: Statistics for Hay on Summer 2015

With a mean of *NEL* (Net Energy content for Lactation) value of 5.5 MJoule out of 51 samples, our case farmer can be quite confident to have bought a hay with a high energy content.

As shown on the top-right of Figure 4.20, additional grouping factors can be selected by the farmer for assessing the quality of the feeds he buys at a more detailed level. Before buying the next Hay stock from the same location, he wants to understand which Hay type provides the highest protein content in order to minimize the amount of protein supplement to give to his cows (a high milk production requires a high protein diet). Figure 4.21 shows the previous result

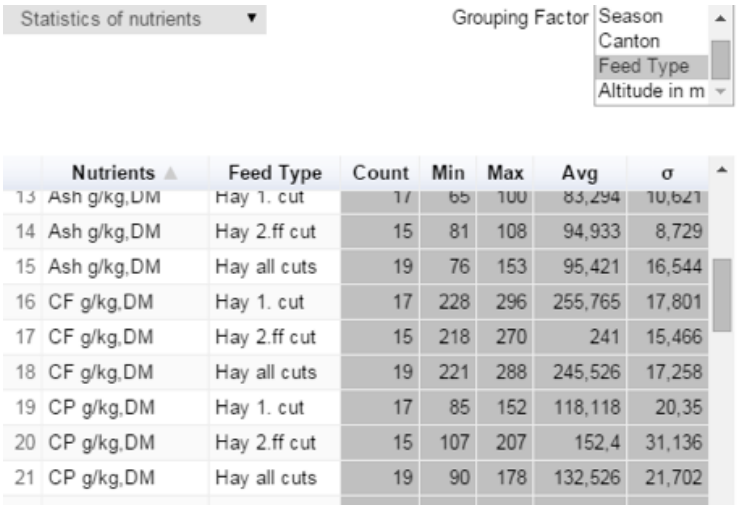


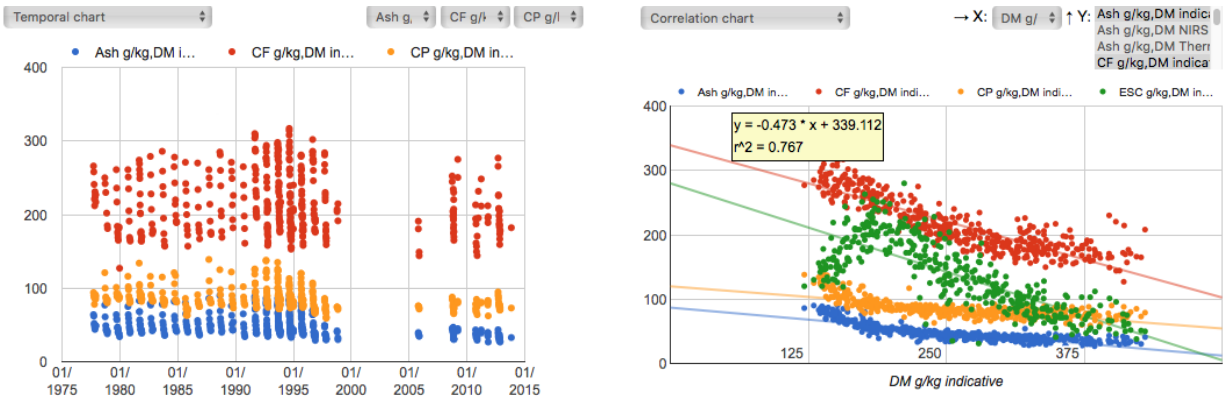
Figure 4.21: The most proteic Hay type during Summer 2015 is Hay of the second cut.

after grouping the result by Feed Type: the farmer should buy Hay of the 2nd cut, since it has the highest *CP* (i.e., Crude Protein) content compared to the one of the 1st cut and the one mixed. Furthermore, based on the current prices, he can evaluate if it's more convenient for him to buy

Hay of the 2nd cut (and give no protein supplement), or to buy cheaper Hay types (and integrate the missing proteins with supplements).

4.9.2 Emergency

Due to a hailstorm, a farmer is forced on a weekend to harvest his whole crop maize field in a premature state before it goes rotten. Without the time (and the means) to chemically analyze the nutritive content of the feed, he would like to have an indication of the quality of his feed. After querying Feedbase.ch, Figure 4.22(a) is obtained. In Figure 4.22(a), the temporal evolution of the nutrients in feed Maize over the last 40 years is shown. To each measurement in the temporal graph, a listener is associated that after a click of the user displays its geographical location in area ③, and its sample information in area ②. For such temporal graph, the farmer selects on top the nutrients whose temporal evolution has to be displayed. In Figure 4.22(a), the farmer has selected three nutrients: Ash (Ash), Crude Fibre (CF), and Crude Protein (CP). The farmer sees that the three nutritive values have a high range of variability (especially CF) which leaves in him a big uncertainty.



(a) Temporal Chart (b) Correlation Chart

Figure 4.22: Evolution of nutrients in feed Maize.

After drying one kg of feed, the farmer knows that the *DM* (Dry Matter) content of his whole crop maize is 250 g. Thus, in area ④ of Feedbase.ch, he switches to a correlation chart, obtaining the graph displayed in Figure 4.22(b). In such a correlation chart, the farmer compares the value of *DM* at axes X, with the content of Ash (Ash), Crude Fiber (*CF*), Crude Protein (*CP*), and Sugar (*ESC*) at axes Y. The farmer can observe that the nutritive values are more stable compared to the

temporal graph. He then uses the linear regressions provided by the system (e.g., in the yellow box it is provided the regression for *CF*, i.e., *CF = -0.473 * DM + 339.112*) and estimates, for *DM*=250, the *CF* value to 220.862 with a coefficient of determination $r^2 = 0.767$.

4.9.3 Similar Regions

During the winter, due to empty stocks, a farmer in the Swiss canton of Thurgau is forced to buy additional Hay from other farms. However, he does not want to unbalance the diet of his animals, i.e., he wants to provide them with feed with similar nutritive values to the one he has grown and given to the animals so far. He therefore uses Feedbase.ch to find the Swiss regions with the most similar Hay to his (target) region.

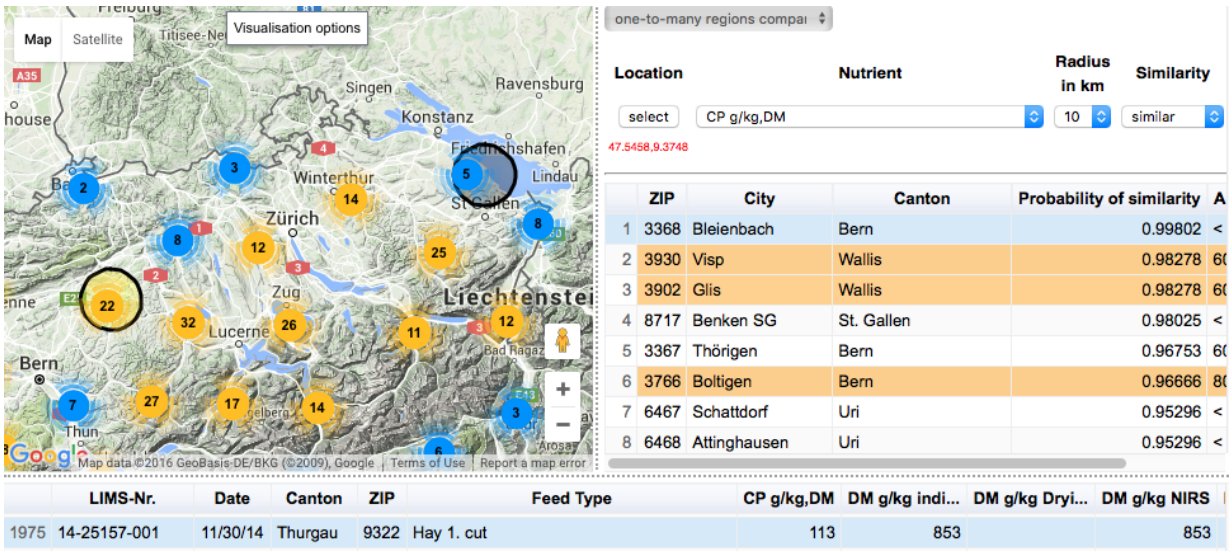


Figure 4.23: The map shows the location of the most similar region (bold right circle) to the target one (bold left circle). The table lists the most similar regions sorted by probability of similarity.

In Figure 4.23 we show that the region having Hay with the most similar content of nutrient *CP* to the the target region (Thurgau 9322, right black circle in the map) is in Canton Bern (left black circle in the map). In the table at the right, the probability of similarity is given, together with the other most similar regions. Similar regions are computed as follows:

1. For each location, all surrounding locations within the selected radius distance are retrieved. Those compose regions.

2. For each region, the average and variance of the measurements of the selected nutrient are computed.

3. For each region that does not intersect the target region, the probability_of_similarity with the target region is computed.

4. The top 30 closest regions are displayed. The regions that are not uniformly distributed according to the shapiro-wilk test are print in orange.

The probability_of_similarity between two regions is computed by first running the *f-test* (which compares the variances of the two regions). If the f-test is larger than 0.05 (i.e., the two regions have similar variance), the *t-test* is run. The t-test compares the averages of two regions that with high probability have the same variance. If the two regions have dissimilar variances, the *welch-t-test* is run. The result is then sorted according to the similarity value returned by the t-test or the welch-t-test.

Due to the high probability of similarity, the farmer requires a price estimation to all farmers in the top 3 locations and buys Hay from the one giving him the best offer.

4.10 Conclusion and Future Work

In this chapter we have presented Feedbase.ch, a system assisting the Swiss animal feeding industry and scientists in the animal feeding process. We have shown the core functionalities and described the lessons learned in implementing our system. We have introduced partial evaluation for efficiently computing data-driven menus in SQL: it re-uses the result of the queries that computed the previous menus, and avoids to redundantly join the dimensions with the fact table. We have introduced derived facts for densifying the sparse data cube of the Swiss Feed Data Warehouse. Derived facts compute nutrients that have not been measured on a given feed sample. We have experimentally shown that an indexed Group- and Selection-Enabled SortMerge is the most robust technique for computing them in SQL since it does not computes redundancies. We have shown how to use materialized views for reducing the runtime in computing distributive aggregation functions in Feedbase.ch by an order of magnitude.

As future work, we are interesting in the computation of derived facts using multidimensional nearest neighbour joins to reduce the imputation error. Our goal is using hashing for reducing the dimensionality of the data, and sorting for the computation of the join itself.

This work has been developed in the context of the Tameus project between the University of Zurich and Agroscope, with funding from the Swiss National Science Foundation.

CHAPTER 5

Conclusion and Future Work

In this thesis, we have introduced techniques for database systems that compute efficient sort-merge computations in the presence of temporal data.

We have included robust support for predicates and groups in sort-merge nearest neighbour joins (NNJs). While current sort-merge implementations fetch blocks storing tuples of different groups multiple times, our group- and selection-enabled NNJ, independent on the number of groups fetches each block at most once. Our approach, independent of the selectivity of the predicate, it does not suffer from index false hits which are major performance bottlenecks in current index-based NNJ solutions. Furthermore, while current approaches restrict the scope of the query optimizer, our group- and selection-enabled NNJ can take advantage of any DBMS optimization on the groups and on the selection predicate. We have described the integration of our solution in the kernel of the open source database system PostgreSQL.

For efficiently managing time intervals in sort-merge computations, we have introduced the Disjoint Interval Partitioning ($\mathcal{DIP}$). $\mathcal{DIP}$ divides an input relation into the smallest number of partitions with non-overlapping tuples. $\mathcal{DIP}$ allows to compute temporal operators (such as joins, anti-joins, aggregations) over the partitions using sort-merge but without having to backtrack to previously scanned tuples. While the number of tuple comparisons done per tuple using

sort-merge is upper-bounded by the number of tuples (i.e., it is quadratic, since backtracking causes many tuples to be rescanned), we make it linear with the number of partitions. By processing multiple partitions at the same time, we reduce the runtime of our worst case scenario, i.e., when many (and, thus, small) partitions exist. Our approach is extremely robust in the presence of long histories of data, since the number of partitions is independent of the length of the history, and there is only a linear dependency between the runtime and the size of partitions.

We have finally described challenges and solutions in implementing a system interfacing a data-warehouse storing temporal feeding data. We have introduced partial evaluation for computing data-driven menus efficiently, without having to redundantly join the fact table with the dimensions. In contrast to state of the art solutions, our approach does not require to restructure the query engine but unlocks efficient query plans already available in current DBMS implementations. We have introduced derived facts for computing nutrients whose value has not been measured in the lab and densifying the sparse data cube of the Swiss Feed Data Warehouse. Derived facts are computed applying chemical formulas after a sequence of NNJs. Finally, for computing nutritive statistics efficiently, we have compared the performance gain obtained using materialized views against directly fetching the measurements from a denormalized or normalized fact table.

Future Work Currently, $\mathcal{DIP}$ needs to be recomputed as soon as the data change or new tuples are added. As a future work, we are interested in incrementally update the $\mathcal{DIP}$ partitions: if a new tuple is added to the database, it is important to minimize the number of operations for finding a non-overlapping partition.

We also plan to efficiently incorporate conditions over non-temporal attributes in temporal operators: while for a temporal equijoin they can be trivially computed on the fly before outputting the result tuples, for anti-joins it becomes complex to generate the leads of the inner tuples since for each value of the non-temporal domain, we need to keep track of the last tuple scanned so far.

For increasing the level of accuracy in computing missing information, we are interested in generalizing the definition and implementation of derived facts with a multidimensional nearest neighbour join that combines hashing (for reducing the dimensionality of the data) and sorting (for computing the join).

Bibliography

[AAO12] Ahmed M. Aly, Walid G. Aref, and Mourad Ouzzani. Spatial queries with two knn predicates. *PVLDB*, 5(11):1100–1111, 2012.

[ABPT01] Mikkel Agesen, Michael H. Böhlen, Lasse Poulsen, and Kristian Torp. A split operator for now-relative bitemporal databases. In *ICDE*, pages 41–50, 2001.

[AEHS+14] Lyublena Antova, Amr El-Helw, Mohamed A. Soliman, Zhongxian Gu, Michalis Petropoulos, and Florian Waas. Optimizing queries over partitioned tables in mpp systems. In *SIGMOD*, pages 373–384, 2014.

[AMDM07] D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. R. Madden. Materialization strategies in a column-oriented dbms. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 466–475, April 2007.

[AMH08] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. Column-stores vs. row-stores: How different are they really? In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 967–980, New York, NY, USA, 2008. ACM.

[APR+98] Lars Arge, Octavian Procopiuc, Sridhar Ramaswamy, Torsten Suel, and Jeffrey Scott Vitter. Scalable sweeping-based spatial join. In *VLDB*, pages 570–581, 1998.

[AR12] Atia M Albhbah and Mick J Ridley. A rule framework for automatic generation of web forms. *International Journal of Computer Theory and Engineering*, 4(4):584, 2012.

[Asi16] Asian acquaculture feed formulation database. http://www.asianaquafeeddatabase.com/, 2016.

[Aus16] Feedplu$4 - feed analysis database. http://dairyinfo.biz/technical-information/farm-business-management/feedplu-ver4-0-feed-analysis-database/, 2016.

[BÖ1] Christian Böhm. The similarity joins: A powerful database primite for high performance data mining. *ICDE, Tutorial*, 2001.

[BBB+12] Monika Boltshauser, Annelies Bracher, Michael H Böhlen, Francesco Cafagna, and Andrej Taliun. Die schweizerische futtermitteldatenbank www.feedbase.ch. *Agrarforschung Schweiz*, 3(2):112–114, 2012.

[BE77] M. W. Blasgen and K. P. Eswaran. Storage and access in relational data bases. *IBM Syst. J.*, 16(4):363–377, 1977.

[BGJ06] Michael Böhlen, Johann Gamper, and Christian S. Jensen. Multi-dimensional aggregation for temporal data. In *EDBT*, pages 257–275, 2006.

[BJS00] Michael H. Böhlen, Christian S. Jensen, and Richard Thomas Snodgrass. Temporal statement modifiers. *ACM Trans. Database Syst.*, 25(4):407–456, December 2000.

[CBB15a] Francesco Cafagna, Michael H. Böhlen, and Annelies Bracher. Nearest neighbour join with groups and predicates. In *Proceedings of the ACM Eighteenth International Workshop on Data Warehousing and OLAP*, DOLAP '15, pages 39–48, New York, NY, USA, 2015. ACM.

[CBB15b] Francesco Cafagna, Michael H. Böhlen, and Annelies Bracher. Nearest neighbour join with groups and predicates. In *DOLAP*, pages 39–48. ACM, 2015.

[CD97] Surajit Chaudhuri and Umeshwar Dayal. An overview of data warehousing and olap technology. *SIGMOD Rec.*, 26(1):65–74, March 1997.

[CGN+14] Bhupesh Chawda, Himanshu Gupta, Sumit Negi, Tanveer A. Faruquie, L. Venkata Subramaniam, and Mukesh K. Mohania. Processing interval joins on map-reduce. In *EDBT*, pages 463–474, 2014.

[CMX13] Alfredo Cuzzocrea, Rim Moussa, and Guandong Xu. *OLAP*: Effectively and Efficiently Supporting Parallel OLAP over Big Data*, pages 38–49. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[cor16] Apache cordova. https://cordova.apache.org, 2016.

[DBG12] Anton Dignös, Michael H. Böhlen, and Johann Gamper. Temporal alignment. In *SIGMOD*, pages 433–444, 2012.

[DBG14] Anton Dignös, Michael H. Böhlen, and Johann Gamper. Overlap interval partition join. In *SIGMOD*, pages 1459–1470, 2014.

[DDL03] C.J. Date, H. Darwen, and N.A. Lorentzos. *Temporal Data and the Relational Model*. Morgan Kaufmann Publishers, pages 77-86, 2003.

[DGSZ03] Vlastislav Dohnal, Claudio Gennaro, Pasquale Savino, and Pavel Zezula. Similarity join in metric spaces. In *ECIR*, pages 452–467, 2003.

[DGZ03] Vlastislav Dohnal, Claudio Gennaro, and Pavel Zezula. Similarity join in metric spaces using ed-index. In *DEXA*, pages 484–493, 2003.

[EHS04] Jost Enderle, Matthias Hampel, and Thomas Seidl. Joining interval data in relational databases. In *SIGMOD*, pages 683–694, 2004.

[ER07] Mohammed M Elsheh and Mick J Ridley. Using database metadata and its semantics to generate automatic and dynamic web entry forms. In *Proceedings of the World Congress on Engineering and Computer Science 2007 WCECS 2007*, 2007.

[Fra16] io - the french feed database. http://www.feedbase.com/, 2016.

[GCB+97] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub totals. *Data Min. Knowl. Discov.*, 1(1):29–53, 1997.

[Ger16] Die datenbank futtermittel der deutsche landwirtschafts-gesellschaft. http://datenbank.futtermittel.net/index.jsp, 2016.

[GHQ95] Ashish Gupta, Venky Harinarayan, and Dallan Quass. Aggregate-query processing in data warehousing environments. In *In Proceedings of the International Conference on Very Large Databases*, pages 358–369, 1995.

[GLJ01] César A. Galindo-Legaria and Milind Joshi. Orthogonal optimization of subqueries and aggregation. In *SIGMOD*, pages 571–581, 2001.

[Goo16] Google developers products. https://developers.google.com/products/, 2016.

[Gra93] Goetz Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–169, June 1993.

[GS91] Himawan Gunadhi and Arie Segev. Query processing algorithms for temporal intersection joins. In *ICDE*, pages 336–344, 1991.

[HLAM06] Stavros Harizopoulos, Velen Liang, Daniel J. Abadi, and Samuel Madden. Performance tradeoffs in read-optimized databases. In *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, pages 487–498. VLDB Endowment, 2006.

[IGN+12] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, and Martin L. Kersten. Monetdb: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 35(1):40–45, 2012.

[JS08] Edwin H. Jacox and Hanan Samet. Metric space similarity joins. *ACM Trans. Database Syst.*, 33(2), 2008.

[Kes08] John N Kesler. Automated generation of dynamic data entry user interface for relational database management systems (p.100), July 15 2008. US Patent 7,401,094.

[KHR+09] Hideaki Kimura, George Huo, Alexander Rasin, Samuel Madden, and Stanley B. Zdonik. Correlation maps: A compressed access method for exploiting soft functional dependencies. *PVLDB*, 2(1):1222–1233, 2009.

[KMV+13] Martin Kaufmann, Amin Amiri Manjili, Panagiotis Vagenas, Peter Michael Fischer, Donald Kossmann, Franz Färber, and Norman May. Timeline index: A unified data

structure for processing queries on temporal data in sap hana. In *SIGMOD*, pages 1173–1184, 2013.

[KPS00] Hans-Peter Kriegel, Marco Pötke, and Thomas Seidl. Managing intervals efficiently in object-relational databases. In *VLDB*, pages 407–418, 2000.

[KR02] Ralph Kimball and Margy Ross. *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling*. John Wiley & Sons, Inc., New York, USA, p.269-271, 2nd edition, 2002.

[KR13a] Ralph Kimball and Margy Ross. *The data warehouse toolkit: The definitive guide to dimensional modeling*. John Wiley & Sons, 2013.

[KR13b] Ralph Kimball and Margy Ross. *The data warehouse toolkit: The definitive guide to dimensional modeling*. John Wiley & Sons, 2013.

[KR13c] Ralph Kimball and Margy Ross. *The data warehouse toolkit: The definitive guide to dimensional modeling*. John Wiley & Sons, 2013.

[KR13d] Ralph Kimball and Margy Ross. *The data warehouse toolkit: The definitive guide to dimensional modeling*. John Wiley & Sons, 2013.

[KS95] N. Kline and R.T. Snodgrass. Computing temporal aggregates. In *ICDE*, pages 222–231, Mar 1995.

[LCC+15] Tirthankar Lahiri, Shasank Chavan, Maria Colgan, Dinesh Das, Amit Ganesh, Mike Gleeson, Sanket Hase, Allison Holloway, Jesse Kamp, Teck-Hua Lee, et al. Oracle database in-memory: A dual format in-memory database. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, pages 1253–1258. IEEE, 2015.

[LGS02] Wei Li, Dengfeng Gao, and Richard Thomas Snodgrass. Skew handling techniques in sort-merge join. In *SIGMOD*, pages 169–180, 2002.

[LM92] T. Y. Cliff Leung and Richard R. Muntz. Temporal query processing and optimization in multiprocessor database machines. In *VLDB*, pages 383–394, 1992.

[LM10] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[LÖ09] Ling Liu and M. Tamer Özsu, editors. *Encyclopedia of Database Systems, p. 671*. Springer US, 2009.

[LR98] Hui Lei and Kenneth A. Ross. Faster joins, self-joins and multi-way joins using join indices. *Data Knowl. Eng.*, 28(3):277–298, December 1998.

[ME92] Priti Mishra and Margaret H. Eich. Join processing in relational databases. *ACM Comput. Surv.*, 24(1):63–113, 1992.

[MFVLI03] B. Moon, I. Fernando Vega Lopez, and V. Immanuel. Efficient algorithms for large-scale temporal aggregation. *TKDE*, 15(3):744–759, May 2003.

[pgn16] Pgnumerics. http://pgnumerics.projects.pgfoundry.org/, 2016.

[PHD16] Danila Piatov, Sven Helmer, and Anton Dignös. An interval join optimized for modern hardware. In *ICDE*, pages 1098–1109, 2016.

[pos16] Postgis 2.2.1. http://www.postgis.net/, 2016.

[PP15] Vasile Purdilă and Ştefan-Gheorghe Pentiuc. Single-scan: a fast star-join query processing algorithm. *Software: Practice and Experience*, 2015.

[SAA10] Yasin N. Silva, Walid G. Aref, and Mohamed H. Ali. The similarity join database operator. In *ICDE*, pages 892–903, 2010.

[SAL+13] Yasin N. Silva, Walid G. Aref, Per-Åke Larson, Spencer Pearson, and Mohamed H. Ali. Similarity queries: their conceptual evaluation, trans- formations, and processing. *VLDB J.*, 22(3):395–420, 2013.

[SAR+14] Mohamed A. Soliman, Lyublena Antova, Venkatesh Raghavan, Amr El-Helw, Zhongxian Gu, Entong Shen, George C. Caragea, Carlos Garcia-Alvarado, Foyzur Rahman, Michalis Petropoulos, Florian Waas, Sivaramakrishnan Narayanan, Konstantinos Krikellas, and Rhonda Baldwin. Orca: a modular query optimizer architecture for big data. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 337–348, 2014.

[SDJL96] Divesh Srivastava, Shaul Dar, H. V. Jagadish, and Alon Y. Levy. Answering queries with aggregation using views. In *Proceedings of the 22th International Conference on Very Large Data Bases*, VLDB '96, pages 318–329, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc.

[SG89] A. Segev and H. Gunadhi. Event-join optimization in temporal relational databases. In *VLDB*, pages 205–215, 1989.

[Sno95] Richard T. Snodgrass, editor. *The TSQL2 Temporal Query Language*. Kluwer, 1995.

[ssa16] Ssa feeds. http://vslp.org/ssafeed/, 2016.

[SSJ94] M.D. Soo, R.T. Snodgrass, and Christian S. Jensen. Efficient evaluation of the valid-time natural join. In *ICDE*, pages 282–292, 1994.

[Str12] Bjarne Stroustrup. Software development for infrastructure. *IEEE Computer*, 45(1):47–58, 2012.

[TBBC12] A. Taliun, M. Böhlen, A. Bracher, and F. Cafagna. A gis-based data analysis platform for analyzing the time-varying quality of animal feed and its impact on the environment. In *iEMSs*, pages 1447–1454, 2012.

[TId15] The time intervals dataset. https://data.gov.uk/dataset/time-intervals, 2015.

[TPC15] TPC. TCP-H benchmark. http://www.tpc.org/tpch/, 2015.

[Ull90] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems: Volume II: The New Technologies*. W. H. Freeman & Co., New York, NY, USA, 1990.

[US16] Feed grains database. http://www.ers.usda.gov/data-products/feed-grains-database.aspx, 2016.

[VLSM05] Ines Fernando Vega Lopez, Richard T. Snodgrass, and Bongki Moon. Spatiotemporal aggregate computation: A survey. *TKDE*, 17(2):271–286, February 2005.

[WQZ+15] Huiju Wang, Xiongpai Qin, Xuan Zhou, Furong Li, Zuoyan Qin, Qing Zhu, and Shan Wang. Efficient query processing framework for big data warehouse: an almost join-free approach. *Frontiers of Computer Science*, 9(2):224–236, 2015.

[YLK10] Bin Yao, Feifei Li, and Piyush Kumar. K nearest neighbor queries and knn-joins in large relational databases (almost) for free. *ICDE*, 0:4–15, 2010.

[YW03] Jun Yang and Jennifer Widom. Incremental computation and maintenance of temporal aggregates. *VLDB J.*, 12(3):262–283, October 2003.